Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Chequeo de tipos eficiente para Path Polymorphism

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Juan Ignacio Edi

Directores: Andrés Viso, Eduardo Bonelli

Buenos Aires, 2015

CHEQUEO DE TIPOS EFICIENTE PARA PATH POLYMORPHISM

Recientemente fue propuesto un sistema de tipos que combina tipos aplicativos, constantes, unión y recursivos, para capturar estáticamente la noción de *Path Polymorphism*. Esto se refiere a la capacidad de operar uniformemente sobre estructuras de datos aplicativas definidas de forma recursiva, utilizando una misma función. La esencia de esta característica se manifiesta en patrones de la forma $x\,y$, que permiten descomponer una estructura de datos arbitraria en las partes que la conforman.

En el presente trabajo estudiamos el problema de type checking para este sistema, realizando esta labor en dos etapas. Primero introducimos un prototipo de implementación que abarca el proceso completo, incluyendo algoritmos para chequeo de equivalencia de tipos y subtipado basados en una caracterización coinductiva de estas relaciones. Se utilizó Haskell para este primer enfoque, para sacar provecho de la naturaleza funcional del problema.

Una vez identificadas las limitaciones del prototipo, se propuso una implementación robusta y eficiente que reduce la complejidad de los pasos clave dentro del proceso, principalmente el chequeo de subtipado. Esta segunda implementación se desarrolló en Scala, utilizando una representación más conveniente para las expresiones de tipos basada en grafos cíclicos.

Palabras clave: Cálculos de Patrones, Chequeo de Tipos, Path Polymorphism, Tipos Recursivos, Operadores ACI.

EFFICIENT TYPE CHECKING FOR PATH POLYMORPHISM

A type system combining type application, constants as types, union types and recursive types has recently been proposed for statically typing *Path Polymorphism*, the ability to define functions that can operate uniformly over recursively specified applicative data structures. A typical pattern such functions resort to is $x\,y$ which decomposes a compound, in other words any applicative tree structure, into its parts.

We study type-checking for this type system in two stages. First we introduce a prototype implementation of the whole process that also involves algorithms for checking type equivalence and subtyping based on coinductive characterizations of those relations. We used Haskell for this first approach, to benefit from the functional nature of the problem.

After identifying the limitations of the prototype, we propose an efficient and robust implementation that reduces complexity of the key part of the process, namely subtyping check. This was done on Scala using a more convenient representation for type expressions based on cyclic graphs.

Keywords: Pattern Calculi, Type Checking, Path Polymorphism, Recursive Types, ACI Operators.

AGRADECIMIENTOS

Comienzo por destacar a Andrés y Eduardo no sólo por su paciencia y el tiempo que dedicaron a que este trabajo exista, sino también por darme una perspectiva mucho más amplia de esta fascinante y chirimbolesca área de estudio. No me olvido tampoco de Emanuel Delgadillo, ni de quien sea que haya determinado que se siente en la misma oficina que nosotros al momento de pulir el algoritmo que presentamos en las páginas que siguen.

Agradezco especialmente a mis padres, cuyas altas espectativas sirvieron de combustible cuando la voluntad propia comenzaba a escasear. Mención distinguida también para la mesa chica de amigos, mis queridísimos compañeros de cursada y todos aquellos que alguna vez me disculparon un faltazo cuando había que cumplir con algún compromiso académico. Espero que el cierre de esta etapa venga seguido de muchos más momentos juntos.

A mi hermana Aisha.

Índice general

1.. Introducción . . . 1
- 1.1. Preliminares . . . 1
  - 1.1.1. Tipos recursivos . . . 2
  - 1.1.2. Coinducción . . . 3
  - 1.1.3. Chequeo de pertenencia a conjuntos coinductivos . . . 4
  - 1.1.4. Calculus of Applicative Patterns . . . 6
- 1.2. Contribución de la tesis . . . 15

2.. Prototipo de implementación . . . 17
- 2.1. Modelado del cálculo . . . 19
- 2.2. Subtipado . . . 21
- 2.3. Equivalencia . . . 22
- 2.4. Análisis de complejidad . . . 23
- 2.5. Discusión . . . 25

3.. Implementación eficiente . . . 27
- 3.1. Cambio de representación . . . 27
- 3.2. Subtipado . . . 30
  - 3.2.1. Algoritmo de verificación . . . 33
- 3.3. Equivalencia . . . 39
  - 3.3.1. Algoritmo de verificación . . . 40
- 3.4. Consideraciones de implementación . . . 42
  - 3.4.1. Representación de conjuntos y diccionarios . . . 42
  - 3.4.2. Optimización del orden de selección de nodos . . . 43
- 3.5. Discusión . . . 44

4.. Conclusiones . . . 47

Apéndice 49

A.. Demostraciones auxiliares . . . 51

1. INTRODUCCIÓN

Los cálculos de patrones son sistemas de reescritura en donde la reducción se basa en **pattern matching** entre funciones y argumentos. En estos sistemas una función está compuesta por distintas ramas que determinan el resultado en base a la estructura del argumento. Dicho mecanismo se utiliza ampliamente en lenguajes de programación como Haskell, Erlang y la familia ML.

Las formalizaciones e implementaciones usuales de pattern matching suelen convivir con dos restricciones. En primer lugar, se trabaja con patrones estáticos, fijos en la definición misma de la función. Por otra parte, la composición del patrón suele estar atada a los constructores de las estructuras de datos esperadas, limitando a un solo tipo los posibles argumentos de la función.

En [JK06, JK09] se introduce el *Pure Pattern Calculus* (abreviado PPC), un cálculo que trata los patrones de forma dinámica. Pueden, por ejemplo, ser generados por el mismo programa o utilizados como parámetros. En este sistema las funciones trabajan sobre datos construidos recursivamente a partir de átomos (que juegan el rol de constructores) y aplicación. Construir de forma uniforme las estructuras de datos permite a su vez representar patrones que descompongan estructuras de forma genérica. Esta característica, denominada **Path Polymorphism**, resulta útil a la hora de definir funciones que recorran estructuralmente expresiones construidas de diversa forma sin atarse a constructores fijos. Podría definirse, por ejemplo, una función que permita computar el tamaño (i.e. número de átomos) de árboles, listas, etc:

$$\begin{array}{rcccl} \mathsf{size} & = & y\,z & \to & \mathsf{size}\,y + \mathsf{size}\,z \\ & \mid & x & \to & 1 \end{array} \tag{1.1}$$

Al aplicar esta función se compara el argumento contra cada uno de los patrones, en el orden en que están definidos. De esta forma, se descompone cualquier estructura compuesta hasta llegar a átomos, independientemente de cuáles sean los constructores concretos que componen la estructura de datos recibida.

Analizar este cálculo desde teoría de tipos resulta particularmente desafiante, dada la naturaleza dinámica de los patrones y la variedad de tipo de parámetros que un mismo patrón puede deconstruir. *Calculus of Applicative Patterns* (abreviado CAP) es un cálculo de patrones inspirado en PPC con soporte de Path Polymorphism restringido a patrones estáticos. En [VBAR15] se presenta un sistema de tipado estático para dicho cálculo que combina tipos aplicativos, uniones idempotentes y tipos recursivos, y se demuestran propiedades fundamentales del funcionamiento del mismo como *Subject Reduction* y *Progress*.

1.1. Preliminares

A continuación se introduce un marco teórico necesario para una mejor comprensión de este trabajo. Esta presentación está fuertemente basada en [Pie02], mientras que el cálculo sobre el cual se basa el desarrollo (CAP) se formaliza en [VBAR15] y [EVB15].

1.1.1. Tipos recursivos

La forma usual de definir tipos de datos nuevos en un lenguaje de programación es expresarlos en términos de constructores que combinan elementos de otros tipos presentes en el sistema. Considerar, por ejemplo, los siguientes tipos algebraicos:

$$\begin{aligned} \texttt{MaybeNat} &= \texttt{Nothing + Just Nat} \\ \texttt{NatList} &= \texttt{Nil + Cons Nat NatList} \end{aligned}$$

En el primer caso, se introduce al sistema un tipo nuevo definido en términos de constructores y un tipo previamente definido. La segunda declaración, por el contrario, presenta problemas si se la interpreta como una definición dirigida, ya que el lado derecho de la igualdad hace referencia al mismo tipo que se está definiendo. Intuitivamente, esta declaración debe entenderse como una ecuación cuya solución será el tipo de las listas de naturales.

Si bien los elementos de este tipo son de tamaño indeterminado, su estructura sigue un patrón sencillo. La ecuación anterior puede interpretarse como un árbol infinito de estructura regular como el siguiente, donde @ denota la aplicación de un constructor a sus argumentos:

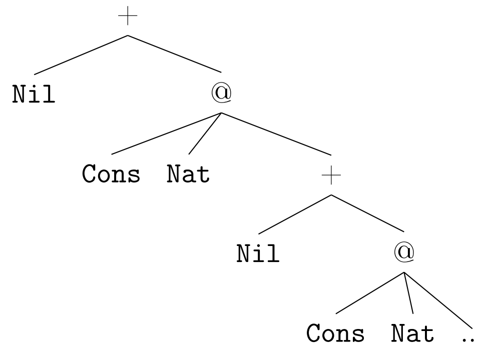


Fig. 1.1: Interpretación del tipo recursivo `NatList` como un árbol infinito

Para manipular formalmente estos tipos infinitos se introduce un operador de punto fijo μ, que nos permite tratar el tipo como una expresión finita que denota la solución a la ecuación en cuestión:

$$\texttt{NatList} \triangleq \mu\alpha.\texttt{Nil} + \texttt{Cons Nat}\ \alpha$$

En la expresión $\mu\alpha.\tau$ el operador μ actúa como ligador de las apariciones de α dentro del cuerpo τ. Esto da lugar a las nociones de variables libres, ligadas y sustituciones clásicas utilizadas en λ-cálculo para las construcciones de la forma $\lambda x.M$. Cabe preguntarse, entonces, cuál es la relación entre una expresión recursiva $\mu\alpha.\tau$ y su desdoblamiento $\{\mu\alpha.\tau/\alpha\}\,\tau$.

Hay dos enfoques para abordar esta pregunta. En un sistema con tipos **iso-recursivos** las expresiones $\mu\alpha.\tau$ y $\{\mu\alpha.\tau/\alpha\}\,\tau$ representan tipos isomorfos pero distintos, y se debe operar explícitamente con operaciones `fold` y `unfold` cuando sea necesario relacionarlos. Esto libera a un programa que trate tipos iso-recursivos de detectar implícitamente las clases de equivalencia de este isomorfismo, dejando al usuario el trabajo de operar sobre los distintos tipos que denotan un mismo árbol.

Alternativamente, en un sistema de tipos **equi-recursivos** tanto la expresión $\mu\alpha.\tau$ como cualquier desdoblamiento de la misma son construcciones sintácticas para referirse al mismo tipo infinito y se consideran equivalentes. Este enfoque resulta más intuitivo, aunque requiere de mayor sofisticación a la hora de definir formalmente las relaciones de equivalencia de tipos y subtipado. Adicionalmente, un programa deberá permitir al usuario tratar indistintamente cualquier desdoblamiento de un mismo tipo, realizando implícitamente las conversiones necesarias para identificar sus distintas representaciones.

En la sección siguiente se presenta la coinducción como marco teórico en que se formalizarán estas nociones y se abordará la implementación de type checkers para tipos equi-recursivos.

1.1.2. Coinducción

Se introduce a continuación la técnica de *coinducción* y su método de prueba asociado en términos de teoría de conjuntos. En secciones siguientes se verá cómo puede utilizarse la herramienta para definir rigurosamente conjuntos de tipos recursivos, así como sus relaciones de equivalencia y subtipado.

Sea U un conjunto fijo que cumple el rol de universo sobre el cual se quiere definir cierto subconjunto.

Definición 1. *Una función $\Phi : \wp(U) \to \wp(U)$ se dice monótona si $A \subseteq B$ implica $\Phi(A) \subseteq \Phi(B)$.*

Utilizaremos de aquí en más Φ para referirnos a funciones monótonas. Al momento de definir un subconjunto de U también las llamaremos *funciones generadoras*, ya que su propósito será deducir la pertenencia al conjunto de ciertos elementos en base a otro subconjunto.

Definición 2. *Dado $X \subseteq U$ se dice que:*

- *X es Φ-denso si $X \subseteq \Phi(X)$.*
- *X es punto fijo de Φ si $X = \Phi(X)$.*

Sabiendo que Φ es monótona, y dado que $\langle \wp(U), \subseteq \rangle$ constituye un reticulado completo, el Teorema del Punto Fijo de Knaster-Tarski [GD03] nos garantiza el siguiente resultado:

Lema 1. *Si $\Phi : \wp(U) \to \wp(U)$ es monótona entonces existe $\nu\Phi$, el mayor punto fijo de Φ, y se define como*

$$\nu\Phi \triangleq \bigcup \{X \in \wp(U) \mid X \subseteq \Phi(X)\}$$

Definición 3 (Conjunto coinductivo denotado por Φ)**.** *Dada $\Phi : \wp(U) \to \wp(U)$ monótona, llamaremos a $\nu\Phi$ el conjunto coinductivo denotado por Φ.*

Se utilizará la notación Φ_X para referirnos a la función generadora del conjunto coinductivo X. Consideremos, a modo de ejemplo, la siguiente función generadora del conjunto E sobre el universo $U = \{a, b, c\}$:

$$\begin{array}{lcl lcl}
\Phi_{\mathsf{E}}(\emptyset) & = & \{c\} & \Phi_{\mathsf{E}}(\{a,b\}) & = & \{c\} \\
\Phi_{\mathsf{E}}(\{a\}) & = & \{c\} & \Phi_{\mathsf{E}}(\{a,c\}) & = & \{b,c\} \\
\Phi_{\mathsf{E}}(\{b\}) & = & \{c\} & \Phi_{\mathsf{E}}(\{b,c\}) & = & \{a,b,c\} \\
\Phi_{\mathsf{E}}(\{c\}) & = & \{b,c\} & \Phi_{\mathsf{E}}(\{a,b,c\}) & = & \{a,b,c\}
\end{array}$$

$$\frac{}{c} \qquad \frac{c}{b} \qquad \frac{b \quad c}{a}$$

Fig. 1.2: Reglas de generación del conjunto E.

Tenemos aquí que los únicos conjuntos Φ_E-densos son $\emptyset, \{c\}, \{b, c\}$ y $\{a, b, c\}$. La unión de los mismos es el conjunto coinductivo denotado por Φ_E, es decir $\mathsf{E} = \nu\Phi_\mathsf{E} = \{a, b, c\}$. Para facilitar la lectura, se utilizarán reglas como las de la Fig. 1.2 para denotar funciones generadoras. En esta notación se interpreta que la presencia de los elementos de la parte superior de la regla en el conjunto de entrada garantiza la pertenencia de los elementos de la parte inferior al conjunto resultante.

Cabe destacar, en general, que $\nu\Phi$ es a su vez un conjunto Φ-denso. Adicionalmente, al estar definido como la unión de todos los conjuntos Φ-densos deberá ser, necesariamente, el mayor de todos.

Lema 2 (Principio de Coinducción)**.** *Si $X \subseteq U$ es Φ-denso, entonces $X \subseteq \nu\Phi$.*

Observar que esta propiedad puede utilizarse como mecanismo de prueba sobre conjuntos coinductivos: para mostrar que un conjunto está incluído en $\nu\Phi$ bastará mostrar que el mismo está contenido en algún conjunto Φ-denso. En la siguiente sección se introducen algoritmos para verificar pertenencia a conjuntos coinductivos basados en este razonamiento: dado un elemento $x \in U$ se busca un conjunto X que contenga a x y sea Φ-denso, de modo de asegurar $x \in \nu\Phi$. Por ejemplo, para probar $b \in \mathsf{E}$ bastaría ver que $b \in \{b, c\}$ y a su vez $\{b, c\} \subseteq \Phi_\mathsf{E}(\{b, c\})$.

1.1.3. Chequeo de pertenencia a conjuntos coinductivos

En esta sección se presenta un método para verificar la pertenencia de un elemento x a un conjunto $\nu\Phi$ definido coinductivamente sobre cierto universo U. Esencialmente, el mecanismo consiste en intentar construir un conjunto Φ-denso que contenga a x. De existir dicho conjunto sabremos por Lem. 2 que está incluído en $\nu\Phi$ y, por lo tanto, $x \in \nu\Phi$.

Decimos que un elemento x es generado por $X \subseteq U$ si $x \in \Phi(X)$. Dado que Φ es monótona, x será generado también por cualquier conjunto X' tal que $X \subseteq X'$. Tiene sentido, entonces, hablar de conjuntos generadores minimales: aquellos que son subconjunto de cualquier otro conjunto generador. El algoritmo de verificación funciona derivando, a partir de la definición de Φ, un conjunto generador de x minimal. Esto es posible para cierta clase de funciones que caracterizamos a continuación.

Definición 4. *Una función generadora Φ se dice **invertible** si para todo $x \in U$*

$$G_x \triangleq \{X \subseteq U \mid x \in \Phi(X)\}$$

es vacío o bien contiene un único conjunto que es subconjunto de todos los otros.

Definición 5. *Se define la función de **soporte** de una función Φ invertible como:*

$$support_\Phi(x) \triangleq \begin{cases} X & si\ X \in G_x\ y\ \forall X' \in G_x.\ X \subseteq X' \\ \uparrow & si\ G_x = \emptyset \end{cases}$$

Cuando el contexto lo permita se utilizará también la función de soporte con dominio en conjuntos:

$$support_\Phi(X) \triangleq \begin{cases} \bigcup_{x \in X} support_\Phi(x) & \textit{si } \forall x \in X.\ support_\Phi(x) \downarrow \\ \uparrow & \textit{en caso contrario} \end{cases}$$

Dado un elemento x, $support_\Phi(x)$ nos dice cuál es el mínimo conjunto de elementos cuya pertenencia a $\nu\Phi$ se deberá verificar para asegurar también la pertenencia de x. Retomando el ejemplo de la sección anterior, tenemos $support_{\Phi_\mathsf{E}}(\{a, b\}) = \{b, c\}$ y $support_{\Phi_\mathsf{E}}(\{c\}) = \emptyset$.

En [Pie02] se sugiere, a partir de las ideas anteriores, el siguiente algoritmo para verificar, dada una función monótona Φ, si un conjunto está incluído en $\nu\Phi$[1]:

```
gfp₀(X) ≜ if support(X) ↑ then
            false
          else if support(X) ⊆ X then
            true
          else gfp₀(X ∪ support(X))
```

Fig. 1.3: Algoritmo de verificación.

Este algoritmo funciona construyendo entre los sucesivos llamados recursivos el conjunto minimal de elementos cuya pertenencia a $\nu\Phi$ se debe verificar. En caso de llegar en algún momento a un conjunto no soportado se concluye que no hay forma de justificar el conjunto original. Por el contrario, si se encuentra que el conjunto contiene todas sus premisas se puede asegurar que es Φ-denso y, por lo tanto, subconjunto de $\nu\Phi$. A continuación se ilustra cómo puede utilizarse este método para verificar la pertenencia de b a E:

$$\begin{aligned} \mathsf{gfp}_0(\{b\}) &= \mathsf{gfp}_0(\{b, c\}) \\ &= \texttt{true} \end{aligned}$$

[Pie02] muestra la corrección de este algoritmo y propone una serie de mejoras incrementales para disminuir la complejidad del mismo. Observar en primer lugar que $support_{\Phi_\mathsf{E}}(b)$ se computa en ambos llamados a gfp_0. Se puede evitar recomputar soportes redundantes distinguiendo entre los elementos ya examinados y aquellos cuyo soporte falta calcular. Estos son, respectivamente, los conjuntos S y X del algoritmo mostrado en la Fig. 1.4.

```
gfp₁(S, X) ≜ if X = ∅ then
               true
             else let x ∈ X in
               if x ∈ S then gfp₁(S, X \ {x})
               else if support(x) ↑ then false
               else gfp₁(S ∪ {x}, (X ∪ support(x)) \ (S ∪ {x}))
```

Fig. 1.4: Algoritmo que evita calcular más de una vez el soporte de cada elemento.

[1] El nombre gpf proviene de *greatest fixed point*, ya que se verifica que un conjunto esté incluído en el mayor punto fijo de la función Φ.

En cada paso, al transferir un elemento de X a S se agregan a X los elementos de su soporte que aún no fueron examinados. De esta forma se busca explorar nuevos elementos necesarios para llegar eventualmente a un conjunto S Φ-denso. Al incorporar esta modificación se tiene el siguiente seguimiento para el ejemplo anterior:

$$\begin{array}{rcl} \mathsf{gfp}_1(\emptyset, \{b\}) & = & \mathsf{gfp}_1(\{b\}, \{c\}) \\ & = & \mathsf{gfp}_1(\{b, c\}, \emptyset) \\ & = & \mathtt{true} \end{array}$$

En efecto, esta versión calcula a lo sumo una vez el soporte de cada elemento. En [Pie02] se sugiere una última modificación sobre este algoritmo, presentada en la Fig. 1.5. Se reemplaza en esta variante el conjunto de trabajo X por un único elemento x a verificar. Al inspeccionar el soporte de x se realizan todos los llamados recursivos correspondientes, devolviendo siempre el conjunto de asunciones aumentado para seguir reutilizando los cálculos ya realizados.

$$\begin{array}{rcl} \mathsf{gfp}(S, x) & \triangleq & \mathtt{if}\ x \in S\ \mathtt{then} \\ & & \quad S \\ & & \mathtt{else\ if}\ \mathsf{support}(x) \uparrow \mathtt{then} \\ & & \quad \mathtt{fail} \\ & & \mathtt{else} \\ & & \quad \mathtt{let}\ \{x_1, ..., x_n\} = \mathsf{support}(x)\ \mathtt{in} \\ & & \quad \mathtt{let}\ S_0 = S \cup \{x\}\ \mathtt{in} \\ & & \quad \mathtt{let}\ S_1 = \mathsf{gfp}(S_0, x_1)\ \mathtt{in} \\ & & \quad ... \\ & & \quad \mathtt{let}\ S_n = \mathsf{gfp}(S_{n-1}, x_n)\ \mathtt{in} \\ & & \quad S_n \end{array}$$

Fig. 1.5: Esquema genérico de verificación de pertenencia a un conjunto coinductivo.

Aquí, para verificar $x \in \nu\Phi$ se computa $\mathsf{gfp}(\emptyset, x)$. Una falla indicará que la pertenencia de x depende de un elemento no soportado. En caso de éxito, se retornará el conjunto Φ-denso minimal que demuestra la pertenencia de x a $\nu\Phi$. Este algoritmo genérico se deberá adaptar según cada Φ, y su corrección y terminación están garantizadas bajo la presunción de que el conjunto de soporte de cada x sea finito.

1.1.4. Calculus of Applicative Patterns

CAP es un cálculo que resulta de restringir PPC a patrones estáticos, focalizándose en *Path Polymorphism* como una de sus características sobresalientes. La esencia de esta propiedad es capturada por patrones de la forma $x\,y$, que permiten descomponer aplicaciones arbitrarias. Combinando esto con una codificación uniforme de estructuras de datos como constructores (c) y aplicaciones binarias de la forma $d\,t$ (donde d es a su vez otra estructura de datos) pueden definirse funciones de consulta genérica como

$$\begin{array}{rlcl} \mathsf{upd} = f \to & (\ \ \mathtt{vl}\,z & \to & \mathtt{vl}\,(f\,z) \\ & \mid\ \ x\,y & \to & (\mathsf{upd}\,f\,x)\,(\mathsf{upd}\,f\,y) \\ & \mid\ \ w & \to & w) \end{array} \tag{1.2}$$

Esta función recorre una estructura arbitraria (ya sea listas, árboles, etc.) aplicando f sólo a aquellos datos prefijados por un constructor `vl`. Cualquier otra aplicación se descompone en sus partes, mediante el patrón $x\,y$, aplicando recursivamente `upd` a cada una de ellas.

El cálculo incorpora la alternativa como un constructor nativo, permitiendo abstraer varios patrones en una misma función. Para esto se asume un conjunto infinito de variables $\mathbb{V}$ y constantes $\mathbb{C}$, distinguiendo en CAP cuatro categorías sintácticas: **patrones** $(p, q, \ldots)$, **términos** $(s, t, \ldots)$, **estructuras de datos** $(d, e, \ldots)$ y **matchable forms** $(m, n, \ldots)$:

$$
\begin{array}{rcll}
p & ::= & x & \text{(matchable)} \\
 & | & \mathsf{c} & \text{(constante)} \\
 & | & p\,p & \text{(compound)} \\
d & ::= & \mathsf{c} & \text{(constante)} \\
 & | & d\,t & \text{(compound)}
\end{array}
\qquad
\begin{array}{rcll}
t & ::= & x & \text{(variable)} \\
 & | & \mathsf{c} & \text{(constante)} \\
 & | & t\,t & \text{(aplicación)} \\
 & | & p \to_\theta t \mid \ldots \mid p \to_\theta t & \text{(abstracción)} \\
m & ::= & d & \text{(estructura de datos)} \\
 & | & p \to_\theta t \mid \ldots \mid p \to_\theta t & \text{(abstracción)}
\end{array}
$$

Los conjuntos de patrones, términos, estructuras de datos y matchable forms se denotan $\mathbb{P}$, $\mathbb{T}$, $\mathbb{D}$ y $\mathbb{M}$, respectivamente. Llamamos **matchables** a las variables presentes en patrones. Se definen de forma tradicional el conjunto de **variables libres** de un término (notación $\mathsf{fv}(t)$), llamadas **matchables libres** ($\mathsf{fm}(p)$) en caso de pertenecer a un patrón. Se utiliza la abreviación $(p_i \to_{\theta_i} s_i)_{i \in 1..n}$ para el término $p_1 \to_{\theta_1} s_1 \mid \ldots \mid p_n \to_{\theta_n} s_n$, donde los θ_i son anotaciones de tipos sobre los matchables del patrón correspondiente. La presencia de matchables en un patrón p_i liga todas las apariciones de la variable en el cuerpo de la rama asociada s_i. Se definen también de la forma esperada **posiciones** en patrones y términos, denotadas con $\pi, \pi', \ldots$ y ϵ para la posición vacía. Escribimos $\mathsf{pos}(s)$ para el conjunto de posiciones de s y $s|_\pi$ para el subtérmino de s en la posición π. Por ejemplo, para $t = \mathsf{c}\,(\mathsf{d}\,\mathsf{e})$ se tiene $\mathsf{pos}(t) = \{1, 21, 22\}$ y $t|_{21} = \mathsf{d}$.

El mecanismo de reducción se basa en una simplificación de la operación de **matching** introducida para PPC al caso estático. Dicha operación puede arrojar tres resultados posibles: éxito, `fail` o `wait`. En el caso de éxito, la misma operación construye la sustitución que instancia el patrón en el argumento. Una **sustitución** se define de la manera usual como una función total que asocia variable a términos, notando con $\{u_1/x_1, \ldots, u_n/x_n\}$ la sustitución que asigna u_i a x_i, con $i \in 1..n$. La unión disjunta de dos resultados de matching se define a continuación:

$$
\begin{array}{rcl}
\mathtt{fail} \uplus o & \triangleq & \mathtt{fail} \\
o \uplus \mathtt{fail} & \triangleq & \mathtt{fail} \\
\sigma_1 \uplus \sigma_2 & \triangleq & \sigma
\end{array}
\qquad
\begin{array}{rcl}
\mathtt{wait} \uplus \sigma & \triangleq & \mathtt{wait} \\
\sigma \uplus \mathtt{wait} & \triangleq & \mathtt{wait} \\
\mathtt{wait} \uplus \mathtt{wait} & \triangleq & \mathtt{wait}
\end{array}
$$

donde o denota cualquier resultado posible y $\sigma_1 \uplus \sigma_2 \triangleq \sigma$ si los dominios de σ_1 y σ_2 son disjuntos. Las reglas de la operación de matching son las siguientes, las cuales se evalúan en el orden presentado:

$$
\begin{array}{lcll}
\{\!\{u/x\}\!\} & \triangleq & \{u/x\} & \\
\{\!\{\mathsf{c}/\mathsf{c}\}\!\} & \triangleq & \{\} & \\
\{\!\{u\,v/p\,q\}\!\} & \triangleq & \{\!\{u/p\}\!\} \uplus \{\!\{v/q\}\!\} & \text{si } u\,v \text{ es un } \textit{matchable form} \\
\{\!\{u/p\}\!\} & \triangleq & \mathtt{fail} & \text{si } u \text{ es un } \textit{matchable form} \\
\{\!\{u/p\}\!\} & \triangleq & \mathtt{wait} &
\end{array}
$$

De esta manera se puede garantizar confluencia del cálculo para la regla (β), dado que se sigue satisfaciendo la *Rigid Matching Condition* [JK09]:

$$\frac{\{\!\{u/p_i\}\!\} = \texttt{fail} \text{ para todo } i < j \quad \{\!\{u/p_j\}\!\} = \sigma_j \quad j \in 1..n}{(p_i \to_{\theta_i} s_i)_{i\in 1..n}\, u \to \sigma_j s_j} \,(\beta)$$

La regla indica que si u es argumento de la abstracción $(p_i \to_{\theta_i} s_i)_{i\in 1..n}$ y matchea con el patrón p_j (produciendo la sustitución σ_j) pero no con los anteriores, entonces se evalúa la rama s_j, sustituyendo las variables libres del patrón en función del resultado del matching.

Sistema de tipos

Presentamos a continuación los μ-types, las expresiones de tipos utilizadas para analizar términos en CAP. Se asumen dados los conjuntos infinitos enumerables $\mathcal{V}_D$ de **variables de datatype** $(\alpha, \beta, \ldots)$, $\mathcal{V}_A$ de **variables de tipo** $(X, Y, \ldots)$ y $\mathcal{C}$ de **constantes de tipo** $(\mathbb{c}, \mathbb{d}, \ldots)$. Se define también $\mathcal{V} \triangleq \mathcal{V}_A \cup \mathcal{V}_D$, cuyos elementos denotaremos con las meta-variables $V, W, \ldots$. A su vez, escribimos $a, b, \ldots$ para elementos en $\mathcal{V} \cup \mathcal{C}$. Los conjuntos $\mathcal{T}_D$ de **μ-datatypes** y $\mathcal{T}$ de **μ-types** se definen inductivamente con la siguiente gramática:

$$\begin{array}{rcll}
D & ::= & \alpha & \text{(variable de datatype)} \\
 & | & \mathbb{c} & \text{(átomo)} \\
 & | & D \mathbin{@} A & \text{(compound)} \\
 & | & D \oplus D & \text{(unión)} \\
 & | & \mu\alpha.D & \text{(recursión)}
\end{array}
\qquad
\begin{array}{rcll}
A & ::= & X & \text{(variable de tipo)} \\
 & | & D & \text{(datatype)} \\
 & | & A \supset A & \text{(funcional)} \\
 & | & A \oplus A & \text{(unión)} \\
 & | & \mu X.A & \text{(recursión)}
\end{array}$$

En esta sintaxis el operador $\oplus$ tiene mayor precedencia que $\supset$, mientras que @ tiene mayor precedencia que $\oplus$. Por lo tanto, $D \mathbin{@} A \oplus A' \supset B$ significa $((D \mathbin{@} A) \oplus A') \supset B$. Escribimos $A \neq \star$, siendo $\star$ un constructor de tipos, para indicar que el símbolo en la raíz de A es distinto de $\star$. Utilizaremos la notación $\mu V.A$ para referirnos indistintamente a expresiones de la forma $\mu\alpha.D$ o $\mu X.A$. Se restringe $\mathcal{T}$ al conjunto de μ-types contractivos: aquellos donde en la expresión $\mu V.A$ la variable V sólo puede ocurrir bajo un constructor de tipos $\supset$ o @. El objetivo de esta restricción es descartar expresiones mal formadas como $\mu V.V$ (cuyo desdoblamiento es equivalente a sí mismo) o $\mu V.V \oplus W$ (que denota un árbol compuesto de infinitas uniones).

Por último, en las Fig. 1.6 y 1.7 se introducen las nociones de equivalencia ($\simeq_\mu$) y subtipado ($\preceq_\mu$) para μ-types contractivos, respectivamente, donde Σ es un contexto de subtipado compuesto por hipótesis de la forma $V \preceq_\mu W$, con $V, W \in \mathcal{V}$.

Se utiliza de aquí en más la notación $\oplus_{i\in 1..n} A_i$ para referirse a la sucesión finita de uniones $A_1 \oplus \ldots \oplus A_n$. Como puede verse en las reglas de equivalencia, el operador $\oplus$ es asociativo, conmutativo e idempotente, por lo que no es necesario distinguir en la expresión anterior a qué posible asociación de expresiones se hace referencia.

Apelando a la noción estándar de contexto de tipado como función parcial de variables en tipos, se introducen dos variantes distintas de juicio de tipado: $\theta \vdash_{\mathsf{p}} p : A$ para patrones y $\Gamma \vdash s : A$ para términos. Notar que los contextos de tipado θ fueron introducidos como anotaciones de tipo para las distintas ramas de las abstracciones en la gramática de términos presentada anteriormente. En la Fig. 1.8 se definen los esquemas que permiten derivar

$$\frac{}{\vdash A \simeq_\mu A}\ (\textsc{e-refl}) \qquad \frac{\vdash A \simeq_\mu B \quad \vdash B \simeq_\mu C}{\vdash A \simeq_\mu C}\ (\textsc{e-trans}) \qquad \frac{\vdash A \simeq_\mu B}{\vdash B \simeq_\mu A}\ (\textsc{e-symm})$$

$$\frac{\vdash D \simeq_\mu D' \quad \vdash A \simeq_\mu A'}{\vdash D \mathbin{@} A \simeq_\mu D' \mathbin{@} A'}\ (\textsc{e-comp}) \qquad \frac{\vdash A \simeq_\mu A' \quad \vdash B \simeq_\mu B'}{\vdash A \supset B \simeq_\mu A' \supset B'}\ (\textsc{e-func})$$

$$\frac{}{\vdash A \oplus A \simeq_\mu A}\ (\textsc{e-idem}) \qquad \frac{}{\vdash A \oplus B \simeq_\mu B \oplus A}\ (\textsc{e-comm})$$

$$\frac{}{\vdash A \oplus (B \oplus C) \simeq_\mu (A \oplus B) \oplus C}\ (\textsc{e-assoc})$$

$$\frac{\vdash A \simeq_\mu A' \quad \vdash B \simeq_\mu B'}{\vdash A \oplus B \simeq_\mu A' \oplus B'}\ (\textsc{e-union}) \qquad \frac{\vdash A \simeq_\mu B}{\vdash \mu V.A \simeq_\mu \mu V.B}\ (\textsc{e-rec})$$

$$\frac{}{\vdash \mu V.A \simeq_\mu \{\mu V.A/V\}\, A}\ (\textsc{e-fold}) \qquad \frac{\vdash A \simeq_\mu \{A/V\}\, B \quad \mu V.B \text{ contractivo}}{\vdash A \simeq_\mu \mu V.B}\ (\textsc{e-contr})$$

Fig. 1.6: Equivalencia para μ-types

juicios de tipado para términos y patrones. Como indica ($\textsc{t-abs}$), para tipar la abstracción $(p_i \to_{\theta_i} s_i)_{i \in 1..n}$ se pide, en principio, que cada p_i sea tipable considerando únicamente θ_i, el contexto de tipado asociado a su rama, y que el dominio de dicho contexto sea exactamente el conjunto de matchables libres del patrón. Como es de esperar, se pide también que el cuerpo de cada rama sea tipable considerando tanto los tipos declarados en las anotaciones como el contexto de tipado Γ. Además, se requiere que la lista $[p_i : A_i]_{i \in 1..n}$ sea *compatible*, noción que se formaliza a continuación.

Compatibilidad

Como ejemplo motivador, retomamos la función `upd` codificada ahora en la sintaxis del cálculo con las anotaciones de tipos correspondientes[2]:

$$\begin{array}{llll} \mathit{upd} \to_{\{\mathit{upd}:A' \supset B'\}} f \to_{\{f:A \supset B\}} (& \mathtt{vl}\, z & \to_{\{z:A\}} & \mathtt{vl}\,(f\, z) \\ & \mid x\, y & \to_{\{x:D, y:E\}} & (\mathit{upd}\, f\, x)\,(\mathit{upd}\, f\, y) \\ & \mid w & \to_{\{w:C\}} & w \qquad) \end{array} \tag{1.3}$$

Si consideramos la aplicación $\mathtt{upd}\,(+1)\,(\mathtt{vl}\,\mathtt{true})$, podemos deducir que tanto A como B deben ser Int. Asumiendo que el término es tipable, se debería entonces tener $D = \mathbb{vl}$ y $E = \mathsf{Bool}$, ya que el tipo del argumento y el de la primera rama difieren. Sin embargo, si reducimos la expresión con el esquema presentado anteriormente obtenemos $\mathtt{vl}\,(\mathtt{true} + 1)$, violando Subject Reduction. El problema surge de que, al buscar preservar la semántica operacional del cálculo no tipado, no se realizan chequeos de tipos al momento de reducir.

[2] Para completar la definición de la función recursiva, tal cual se la presenta en el ejemplo 1.1, basta aplicar al término un operador de punto fijo.

$$\frac{}{\Sigma \vdash A \preceq_\mu A}\ (\text{S-REFL}) \qquad \frac{}{\Sigma, V \preceq_\mu W \vdash V \preceq_\mu W}\ (\text{S-HYP}) \qquad \frac{\vdash A \simeq_\mu B}{\Sigma \vdash A \preceq_\mu B}\ (\text{S-EQ})$$

$$\frac{\Sigma \vdash A \preceq_\mu B \quad \Sigma \vdash B \preceq_\mu C}{\Sigma \vdash A \preceq_\mu C}\ (\text{S-TRANS}) \qquad \frac{\Sigma \vdash D \preceq_\mu D' \quad \Sigma \vdash A \preceq_\mu A'}{\Sigma \vdash D \mathbin{@} A \preceq_\mu D' \mathbin{@} A'}\ (\text{S-COMP})$$

$$\frac{\Sigma \vdash A \preceq_\mu A' \quad \Sigma \vdash B \preceq_\mu B'}{\Sigma \vdash A' \supset B \preceq_\mu A \supset B'}\ (\text{S-FUNC}) \qquad \frac{\Sigma \vdash A \preceq_\mu C \quad \Sigma \vdash B \preceq_\mu C}{\Sigma \vdash A \oplus B \preceq_\mu C}\ (\text{S-UNION-L})$$

$$\frac{\Sigma \vdash A \preceq_\mu B}{\Sigma \vdash A \preceq_\mu B \oplus C}\ (\text{S-UNION-R1}) \qquad \frac{\Sigma \vdash A \preceq_\mu C}{\Sigma \vdash A \preceq_\mu B \oplus C}\ (\text{S-UNION-R2})$$

$$\frac{\Sigma, V \preceq_\mu W \vdash A \preceq_\mu B \quad W \notin \mathsf{fv}(A) \quad V \notin \mathsf{fv}(B)}{\Sigma \vdash \mu V.A \preceq_\mu \mu W.B}\ (\text{S-REC})$$

Fig. 1.7: Subtipado fuerte para μ-types

Para garantizar que estas situaciones no se presentan se introduce la noción de *compatibilidad.*

Ésta busca garantizar que un argumento válido para la segunda rama (desde el punto de vista del tipado) no sea capturado por la primera como consecuencia del matching. Un análisis formal requiere analizar las posiciones de mismatch entre los distintos patrones para poder detectar potenciales solapamientos entre sus conjuntos de argumentos y, de esa forma, restringir los tipos a asignar. Introducimos primero algunas definiciones auxiliares para formalizar esta noción.

Definición 6. *Diremos que un patrón p* ***subsume*** *a otro patrón q, notado $p \lhd q$, si existe una sustitución σ tal que $\sigma p = q$.*

Definición 7.
Las ***posiciones maximales*** *de un conjunto de posiciones P son:*

$$\mathsf{maxpos}(P) \triangleq \left\{\pi \in P \mid \nexists \pi' \in P.\pi' = \pi\pi'' \wedge \pi'' \neq \epsilon\right\}$$

Las ***posiciones de mismatch*** *entre dos patrones son:*

$$\mathsf{mmpos}(p, q) \triangleq \{\pi \mid \pi \in \mathsf{maxpos}(\mathsf{pos}(p) \cap \mathsf{pos}(q)) \wedge p|_\pi \ntriangleleft q|_\pi\}$$

Definición 8. *Dados el juicio derivable $\theta \vdash_{\mathsf{p}} p : A$ y $\pi \in \mathsf{pos}(p)$, decimos que A* ***admite el símbolo*** *$\odot$ (con $\odot \in \mathcal{V} \cup \mathcal{C} \cup \{@, \supset\}$) en la posición π si y sólo si $\odot \in A\|_\pi$, donde:*

$$\begin{array}{rcll}
a\|_\epsilon & \triangleq & \{a\} & \\
(A_1 \star A_2)\|_\epsilon & \triangleq & \{\star\}, & \star \in \{@, \supset\} \\
(A_1 \star A_2)\|_{i\pi} & \triangleq & A_i\|_\pi, & \star \in \{@, \supset\}, i \in \{1, 2\} \\
(A_1 \oplus A_2)\|_\pi & \triangleq & A_1\|_\pi \cup A_2\|_\pi & \\
(\mu V.A')\|_\pi & \triangleq & (\{\mu V.A'/V\}\, A')\|_\pi &
\end{array}$$

Patrones

$$\frac{\theta(x) = A}{\theta \vdash_{\mathsf{p}} x : A}\ (\text{P-MATCH}) \qquad \frac{}{\theta \vdash_{\mathsf{p}} \mathsf{c} : \mathbb{c}}\ (\text{P-CONST}) \qquad \frac{\theta \vdash_{\mathsf{p}} p : D \quad \theta \vdash_{\mathsf{p}} q : A}{\theta \vdash_{\mathsf{p}} p\,q : D \mathbin{@} A}\ (\text{P-COMP})$$

Términos

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}\ (\text{T-VAR}) \qquad \frac{}{\Gamma \vdash \mathsf{c} : \mathbb{c}}\ (\text{T-CONST}) \qquad \frac{\Gamma \vdash r : D \quad \Gamma \vdash u : A}{\Gamma \vdash r\,u : D \mathbin{@} A}\ (\text{T-COMP})$$

$$\frac{\begin{array}{c}[p_i : A_i]_{i\in 1..n} \text{ compatible} \\ (\theta_i \vdash_{\mathsf{p}} p_i : A_i)_{i\in 1..n} \quad (\mathsf{dom}\,(\theta_i) = \mathsf{fm}(p_i))_{i\in 1..n} \quad (\Gamma, \theta_i \vdash s_i : B)_{i\in 1..n}\end{array}}{\Gamma \vdash (p_i \to_{\theta_i} s_i)_{i\in 1..n} : \bigoplus_{i\in 1..n} A_i \supset B}\ (\text{T-ABS})$$

$$\frac{\Gamma \vdash r : \bigoplus_{i\in 1..n} A_i \supset B \quad \Gamma \vdash u : A_k \quad k \in 1..n}{\Gamma \vdash r\,u : B}\ (\text{T-APP}) \qquad \frac{\Gamma \vdash s : A \quad \vdash A \preceq_\mu A'}{\Gamma \vdash s : A'}\ (\text{T-SUBS})$$

Fig. 1.8: Reglas de tipado para patrones y términos

Si se considera el siguiente predicado para detectar los posibles solapamientos mencionados anteriormente

$$\mathcal{P}_{\mathsf{comp}}(p : A, q : B) \triangleq \forall \pi \in \mathsf{mmpos}(p, q)\,.A\|_\pi \cap B\|_\pi \neq \varnothing$$

la noción de compatibilidad entre patrones puede definirse del siguiente modo:

Definición 9. *Decimos que un patrón $p : A$ es **compatible** con $q : B$ si*

$$\mathcal{P}_{\mathsf{comp}}(p : A, q : B) \implies B \preceq_\mu A$$

Una secuencia de patrones $[p_i : A_i]_{i\in 1..n}$ se dice compatible si $p_j : A_j$ es compatible con $p_k : A_k$ para todo $j, k \in 1..n$ con $j < k$.

Previamente se mencionó que de ser tipable el término $\mathsf{upd}\,(+1)\,(\mathtt{vl}\,\mathtt{true})$ debían asignarse los siguientes tipos a los patrones de las primeras ramas internas de *upd*:

$$\begin{array}{rcl}\mathtt{vl}\,z & : & \mathbb{vl} \mathbin{@} \mathsf{Int} \\ x\,y & : & \mathbb{vl} \mathbin{@} \mathsf{Bool}\end{array}$$

Observar que en este caso la única posición de mismatch es la del subtérmino izquierdo de ambos patrones, en donde vale $\mathtt{vl}\,z\|_1 \cap x\,y\|_1 = \{\mathbb{vl}\}$. Al cumplirse $\mathcal{P}_{\mathsf{comp}}$ se deberá imponer la restricción de subtipado sobre los tipos asignados. Dado que $\mathbb{vl} \mathbin{@} \mathsf{Bool} \not\preceq_\mu \mathbb{vl} \mathbin{@} \mathsf{Int}$ se declara ambas ramas incompatibles, produciendo una falla de tipado y evitando así la captura del argumento $\mathtt{vl}\,\mathtt{true}$ por la primera rama.

Formulación dirigida por sintaxis

Los esquemas de tipado presentados en la Fig. 1.8 no resultan muy convenientes al abordar el problema de chequeo de tipos. Dado un juicio $\Gamma \vdash s : A$, sería deseable que

$$\frac{\Gamma(x) = A}{\Gamma \mapsto x : A}\,(\text{T-VAR-AL}) \qquad \frac{}{\Gamma \mapsto \mathsf{c} : \mathbb{c}}\,(\text{T-CONST-AL}) \qquad \frac{\Gamma \mapsto r : D \quad \Gamma \mapsto u : A}{\Gamma \mapsto r\,u : D \mathbin{@} A}\,(\text{T-COMP-AL})$$

$$\frac{\begin{array}{c}[p_i : A_i]_{i \in 1..n} \text{ compatible} \\ (\theta_i \vdash_{\mathsf{p}} p_i : A_i)_{i \in 1..n} \quad (\mathsf{dom}\,(\theta_i) = \mathsf{fm}(p_i))_{i \in 1..n} \quad (\Gamma, \theta_i \mapsto s_i : B_i)_{i \in 1..n}\end{array}}{\Gamma \mapsto (p_i \to_{\theta_i} s_i)_{i \in 1..n} : \bigoplus_{i \in 1..n} A_i \supset \bigoplus_{i \in 1..n} B_i}\,(\text{T-ABS-AL})$$

$$\frac{\begin{array}{l}\Gamma \mapsto r : A \quad A \simeq_{\mu} \oplus_{i \in 1..n}(A_i \supset B_i) \quad A_i \neq \oplus \\ \Gamma \mapsto u : C \quad (\vdash C \preceq_{\mu} A_i)_{i \in 1..n}\end{array}}{\Gamma \mapsto r\,u : \bigoplus_{i \in 1..n} B_i}\,(\text{T-APP-AL})$$

Fig. 1.9: Reglas de tipado para términos

bastase con observar la estructura de s para determinar qué regla utilizar en el primer paso de derivación. Esta formulación, sin embargo, no es dirigida por sintaxis dado que (T-SUBS) puede en principio ser aplicada sobre cualquier expresión. Más aún, en los casos en que es necesario aplicar esta regla, sería responsabilidad del programa encargado de verificar el juicio *adivinar* cuál es el tipo A a utilizar. Para obtener una formulación dirigida por sintaxis se debe, entonces, descartar (T-SUBS), y modificar las reglas que correspondan para considerar subtipado. En [EVB15] se introducen los esquemas dirigidos por sintaxis presentados en la Fig. 1.9, los cuales cumplen con la siguiente propiedad de corrección:

Proposición 1.

1. *Si* $\Gamma \mapsto s : A$, *entonces* $\Gamma \vdash s : A$.
2. *Si* $\Gamma \vdash s : A$, *entonces* $\exists A'$ *tal que* $A' \preceq_{\mu} A$ *y* $\Gamma \mapsto s : A'$.

Observar que la regla de transitividad (S-TRANS) introduce un problema similar para verificar la noción de subtipado $\preceq_{\mu}$. Esto motiva la formulación de la relación coinductiva $\preceq_{\vec{\mu}}$, cuyas reglas son invertibles (ver Fig. 1.10). Esta relación se define formalmente como $\nu\Phi_{\preceq_{\vec{\mu}}}$:

$$\begin{array}{rcl}
\Phi_{\preceq_{\vec{\mu}}}(\mathcal{S}) & = & \{\langle a, a\rangle \mid a \in \mathcal{V} \cup \mathcal{C}\} \\
& \cup & \{\langle D \mathbin{@} A, D' \mathbin{@} A'\rangle \mid \langle D, D'\rangle, \langle A, A'\rangle \in \mathcal{S}\} \\
& \cup & \{\langle A \supset B, A' \supset B'\rangle \mid \langle A', A\rangle, \langle B, B'\rangle \in \mathcal{S}\} \\
& \cup & \{\langle \mu V.A, B\rangle \mid \langle \{\mu V.A/V\}\,A, B\rangle \in \mathcal{S}\} \\
& \cup & \{\langle A, \mu W.B\rangle \mid \langle A, \{\mu W.B/W\}\,B\rangle \in \mathcal{S}, A \neq \mu\} \\
& \cup & \{\langle \oplus_{i \in 1..n} A_i, B\rangle \mid \langle A_i, B\rangle \in \mathcal{S} \text{ para todo } i \in 1..n, n > 1, B \neq \mu, A_i \neq \oplus\} \\
& \cup & \{\langle A, \oplus_{j \in 1..m} B_j\rangle \mid \langle A, B_k\rangle \in \mathcal{S} \text{ para algún } k \in 1..m, m > 1, A \neq \mu, \oplus, B_j \neq \oplus\}
\end{array}$$

Esta axiomatización efectivamente captura la misma noción de subtipado sobre μ-types que la relación inductiva $\preceq_{\mu}$ presentada originalmente.

Proposición 2. $A \preceq_{\vec{\mu}} B$ *si y sólo si* $A \preceq_{\mu} B$.

Del mismo modo, surge la necesidad de reformular la relación de equivalencia $\simeq_{\mu}$, cuya alternativa dirigida por sintaxis $\simeq_{\vec{\mu}}$ se presenta en la Fig. 1.11. Para esto se introduce la

$$\frac{}{a \preceq_{\vec{\mu}} a} \text{(S-REFL-AL)}$$

$$\frac{D \preceq_{\vec{\mu}} D' \quad A \preceq_{\vec{\mu}} A'}{D \mathbin{@} A \preceq_{\vec{\mu}} D' \mathbin{@} A'} \text{(S-COMP-AL)} \qquad \frac{A' \preceq_{\vec{\mu}} A \quad B \preceq_{\vec{\mu}} B'}{A \supset B \preceq_{\vec{\mu}} A' \supset B'} \text{(S-FUNC-AL)}$$

$$\frac{\{\mu V.A/V\}\, A \preceq_{\vec{\mu}} B}{\mu V.A \preceq_{\vec{\mu}} B} \text{(S-REC-L-AL)} \qquad \frac{A \preceq_{\vec{\mu}} \{\mu W.B/W\}\, B \quad A \neq \mu}{A \preceq_{\vec{\mu}} \mu W.B} \text{(S-REC-R-AL)}$$

$$\frac{A_i \preceq_{\vec{\mu}} B \text{ para todo } i \in 1..n \quad n > 1 \quad B \neq \mu \quad A_i \neq \oplus}{\bigoplus_{i \in 1..n} A_i \preceq_{\vec{\mu}} B} \text{(S-UNION-L-AL)}$$

$$\frac{A \preceq_{\vec{\mu}} B_k \text{ para algún } k \in 1..m \quad m > 1 \quad A \neq \mu, \oplus \quad B_j \neq \oplus}{A \preceq_{\vec{\mu}} \bigoplus_{j \in 1..m} B_j} \text{(S-UNION-R-AL)}$$

Fig. 1.10: Axiomatización coinductiva de subtipado para μ-types contractivos

siguiente noción de contexto (meta-variables $\mathscr{C}, \mathscr{D}$):

$$A_1 \oplus \ldots A_{i-1} \oplus \Box \oplus A_{i+1} \oplus \ldots \oplus A_n$$

con $A_j \neq \oplus$ para todo $j \in 1..n \setminus i$ y $A_l \neq \mu$ para todo $l \in 1..i-1$[3]. Formalmente, se define $\simeq_{\vec{\mu}}$ como $\nu\Phi_{\simeq_{\vec{\mu}}}$, donde:

$$\begin{array}{rcl}
\Phi_{\simeq_{\vec{\mu}}}(\mathcal{S}) & = & \{\langle a, a\rangle \mid a \in \mathcal{V} \cup \mathcal{C}\} \\
& \cup & \{\langle D \mathbin{@} A, D' \mathbin{@} A'\rangle \mid \langle D, D'\rangle, \langle A, A'\rangle \in \mathcal{S}\} \\
& \cup & \{\langle A \supset B, A' \supset B'\rangle \mid \langle A, A'\rangle, \langle B, B'\rangle \in \mathcal{S}\} \\
& \cup & \{\langle \mathscr{C}[\mu V.A], B\rangle \mid \langle \mathscr{C}[\{\mu V.A/V\}\, A], B\rangle \in \mathcal{S}\} \\
& \cup & \{\langle A, \mathscr{D}[\mu W.B]\rangle \mid \langle A, \mathscr{D}[\{\mu W.B/W\}\, B]\rangle \in \mathcal{S}, A \neq \mathscr{C}[\mu V.C]\} \\
& \cup & \{\langle \oplus_{i \in 1..n} A_i, \oplus_{j \in 1..m} B_j\rangle \mid n + m > 2, A_i, B_j \neq \mu, \oplus \\
& & \quad \exists f : 1..n \to 1..m \text{ tal que } \langle A_i, B_{f(i)}\rangle \in \mathcal{S}, \\
& & \quad \exists g : 1..m \to 1..n \text{ tal que } \langle A_{g(j)}, B_j\rangle \in \mathcal{S}\}
\end{array}$$

Al igual que en el caso de subtipado, esta relación captura la misma noción que su contraparte inductiva.

Proposición 3. *$A \simeq_{\vec{\mu}} B$ si y sólo si $A \simeq_{\mu} B$.*

Interpretación de μ-types como árboles

Alternativamente, en [VBAR15] se introduce una interpretación semántica de los tipos contractivos en el espacio métrico completo $\mathfrak{T}$ de **árboles infinitos** ($\mathcal{A}, \mathcal{B}, \ldots$) [Cou83]. El alfabeto usado para esta construcción es $\mathfrak{L} \triangleq \{a^0 \mid a \in \mathcal{V} \cup \mathcal{C}\} \cup \{@^2, \supset^2, \oplus^2\}$, por lo que se obtiene un conjunto de árboles binarios. Se tiene particular interés por el subconjunto $\mathfrak{T}^{reg}$ de **árboles regulares**, aquellos a los cuales se mapean los μ-types mediante la siguiente función de traducción.

[3] Notar que, en particular, $\mathscr{C}$ puede tomar la forma $\Box$.

$$\frac{}{a \simeq_{\vec{\mu}} a}\ (\text{E-REFL-AL})$$

$$\frac{D \simeq_{\vec{\mu}} D' \quad A \simeq_{\vec{\mu}} A'}{D \mathbin{@} A \simeq_{\vec{\mu}} D' \mathbin{@} A'}\ (\text{E-COMP-AL}) \qquad \frac{A \simeq_{\vec{\mu}} A' \quad B \simeq_{\vec{\mu}} B'}{A \supset B \simeq_{\vec{\mu}} A' \supset B'}\ (\text{E-FUNC-AL})$$

$$\frac{\mathscr{C}[\{\mu V.A/V\}\, A] \simeq_{\vec{\mu}} B}{\mathscr{C}[\mu V.A] \simeq_{\vec{\mu}} B}\ (\text{E-REC-L-AL}) \qquad \frac{A \simeq_{\vec{\mu}} \mathscr{D}[\{\mu W.B/W\}\, B] \quad A \neq \mathscr{C}[\mu V.C]}{A \simeq_{\vec{\mu}} \mathscr{D}[\mu W.B]}\ (\text{E-REC-R-AL})$$

$$\frac{\begin{array}{ll} A_i \simeq_{\vec{\mu}} B_{f(i)} & f : 1..n \to 1..m \\ A_{g(j)} \simeq_{\vec{\mu}} B_j & g : 1..m \to 1..n \end{array} \quad A_i, B_j \neq \mu, \oplus \quad n+m > 2}{\bigoplus_{i \in 1..n} A_i \simeq_{\vec{\mu}} \bigoplus_{j \in 1..m} B_j}\ (\text{E-UNION-AL})$$

Fig. 1.11: Axiomatización coinductiva de equivalencia para μ-types contractivos

$$\frac{}{a \preceq_{\mathfrak{T}} a}\ (\text{S-REFL-T})$$

$$\frac{\mathcal{D} \preceq_{\mathfrak{T}} \mathcal{D}' \quad \mathcal{A} \preceq_{\mathfrak{T}} \mathcal{A}'}{\mathcal{D} \mathbin{@} \mathcal{A} \preceq_{\mathfrak{T}} \mathcal{D}' \mathbin{@} \mathcal{A}'}\ (\text{S-COMP-T}) \qquad \frac{\mathcal{A}' \preceq_{\mathfrak{T}} \mathcal{A} \quad \mathcal{B} \preceq_{\mathfrak{T}} \mathcal{B}'}{\mathcal{A} \supset \mathcal{B} \preceq_{\mathfrak{T}} \mathcal{A}' \supset \mathcal{B}'}\ (\text{S-FUNC-T})$$

$$\frac{\mathcal{A}_i \preceq_{\mathfrak{T}} \mathcal{B}_{f(i)} \quad f : 1..n \to 1..m \quad \mathcal{A}_i, \mathcal{B}_j \neq \oplus \quad n+m > 2}{\bigoplus_{i \in 1..n} \mathcal{A}_i \preceq_{\mathfrak{T}} \bigoplus_{j \in 1..m} \mathcal{B}_j}\ (\text{S-UNION-T})$$

Fig. 1.12: Relación de subtipado para árboles infinitos.

Definición 10. *La función* $[\![\cdot]\!]^{\mathfrak{T}} : \mathcal{T} \to \mathfrak{T}^{reg}$ *se define como:*

$$\begin{array}{rcll} [\![a]\!]^{\mathfrak{T}}(\epsilon) & \triangleq & a & \\ [\![A_1 \star A_2]\!]^{\mathfrak{T}}(\epsilon) & \triangleq & \star & \star \in \{@, \supset, \oplus\} \\ [\![A_1 \star A_2]\!]^{\mathfrak{T}}(i\pi) & \triangleq & [\![A_i]\!]^{\mathfrak{T}}(\pi) & \star \in \{@, \supset, \oplus\}, i \in \{1,2\} \\ [\![\mu V.A]\!]^{\mathfrak{T}}(\pi) & \triangleq & [\![\{\mu V.A/V\}\, A]\!]^{\mathfrak{T}}(\pi) & \end{array}$$

Cabe destacar que el conjunto $\mathfrak{T}$ (y consecuentemente $\mathfrak{T}^{reg}$) no incluye árboles que contengan ramas infinitas compuestas únicamente por uniones. Esto garantiza que todo árbol regular tiene una representación finita como μ-type contractivo.

Pueden definirse también nociones adecuadas de subtipado y equivalencia de manera coinductiva (ver Fig. 1.12 y 1.13, resp.), cuyas funciones generadoras son:

$$\begin{array}{rcl} \Phi_{\preceq_{\mathfrak{T}}}(\mathcal{S}) & = & \{\langle a, a\rangle \mid a \in \mathcal{V} \cup \mathcal{C}\} \\ & \cup & \{\langle \mathcal{D} \mathbin{@} \mathcal{A}, \mathcal{D}' \mathbin{@} \mathcal{A}'\rangle \mid \langle \mathcal{D}, \mathcal{D}'\rangle, \langle \mathcal{A}, \mathcal{A}'\rangle \in \mathcal{S}\} \\ & \cup & \{\langle \mathcal{A} \supset \mathcal{B}, \mathcal{A}' \supset \mathcal{B}'\rangle \mid \langle \mathcal{A}', \mathcal{A}\rangle, \langle \mathcal{B}, \mathcal{B}'\rangle \in \mathcal{S}\} \\ & \cup & \{\langle \oplus_{i \in 1..n} \mathcal{A}_i, \oplus_{j \in 1..m} \mathcal{B}_j\rangle \mid \mathcal{A}_i, \mathcal{B}_j \neq \oplus, n+m > 2 \\ & & \quad \exists f : 1..n \to 1..m \text{ tal que } \langle \mathcal{A}_i, \mathcal{B}_{f(i)}\rangle \in \mathcal{S}\} \end{array}$$

$$\frac{}{a \simeq_{\mathfrak{T}} a}\ (\text{E-REFL-T})$$

$$\frac{\mathcal{D} \simeq_{\mathfrak{T}} \mathcal{D}' \quad \mathcal{A} \simeq_{\mathfrak{T}} \mathcal{A}'}{\mathcal{D} @ \mathcal{A} \simeq_{\mathfrak{T}} \mathcal{D}' @ \mathcal{A}'}\ (\text{E-COMP-T}) \qquad \frac{\mathcal{A} \simeq_{\mathfrak{T}} \mathcal{A}' \quad \mathcal{B} \simeq_{\mathfrak{T}} \mathcal{B}'}{\mathcal{A} \supset \mathcal{B} \simeq_{\mathfrak{T}} \mathcal{A}' \supset \mathcal{B}'}\ (\text{E-FUNC-T})$$

$$\frac{\begin{array}{ll}\mathcal{A}_i \simeq_{\mathfrak{T}} \mathcal{B}_{f(i)} & f : 1..n \to 1..m \\ \mathcal{A}_{g(j)} \simeq_{\mathfrak{T}} \mathcal{B}_j & g : 1..m \to 1..n\end{array} \quad \mathcal{A}_i, \mathcal{B}_j \neq \oplus \quad n+m>2}{\bigoplus_{i\in 1..n}\mathcal{A}_i \simeq_{\mathfrak{T}} \bigoplus_{j \in 1..m}\mathcal{B}_j}\ (\text{E-UNION-T})$$

Fig. 1.13: Relación de equivalencia para árboles infinitos.

$$\begin{array}{rcl}\Phi_{\simeq_{\mathfrak{T}}}(\mathcal{S}) & = & \{\langle a, a\rangle \mid a \in \mathcal{V} \cup \mathcal{C}\} \\ & \cup & \{\langle \mathcal{D} @ \mathcal{A}, \mathcal{D}' @ \mathcal{A}'\rangle \mid \langle \mathcal{D}, \mathcal{D}'\rangle, \langle \mathcal{A}, \mathcal{A}'\rangle \in \mathcal{S}\} \\ & \cup & \{\langle \mathcal{A} \supset \mathcal{B}, \mathcal{A}' \supset \mathcal{B}'\rangle \mid \langle \mathcal{A}, \mathcal{A}'\rangle, \langle \mathcal{B}, \mathcal{B}'\rangle \in \mathcal{S}\} \\ & \cup & \{\langle \oplus_{i\in 1..n}\mathcal{A}_i, \oplus_{j\in 1..m}\mathcal{B}_j\rangle \mid \mathcal{A}_i, \mathcal{B}_j \neq \oplus, n+m>2 \\ & & \quad \exists f : 1..n \to 1..m \text{ tal que } \langle \mathcal{A}_i, \mathcal{B}_{f(i)}\rangle \in \mathcal{S}, \\ & & \quad \exists g : 1..m \to 1..n \text{ tal que } \langle \mathcal{A}_{g(j)}, \mathcal{B}_j\rangle \in \mathcal{S}\}\end{array}$$

Estas relaciones se comportan de la manera esperada respecto a las nociones de subtipado y equivalencia para μ-types, vía la interpretación $[\![\cdot]\!]^{\mathfrak{T}}$.

Proposición 4. *$A \preceq_\mu B$ si y sólo si $[\![A]\!]^{\mathfrak{T}} \preceq_{\mathfrak{T}} [\![B]\!]^{\mathfrak{T}}$.*

Proposición 5. *$A \simeq_\mu B$ si y sólo si $[\![A]\!]^{\mathfrak{T}} \simeq_{\mathfrak{T}} [\![B]\!]^{\mathfrak{T}}$.*

1.2. Contribución de la tesis

Se presenta a continuación un algoritmo eficiente de chequeo de tipos para CAP y su implementación correspondiente. En una primera etapa se desarrolló en Haskell un prototipo basado fuertemente en las formulaciones del sistema introducidas hasta el momento, que fue de utilidad para identificar las dificultades del problema.

Con el objetivo principal de mejorar el orden de complejidad de la solución y poniendo el foco en lograr una implementación de buen desempeño práctico se implementó una segunda versión, que motivó la introducción de una noción alternativa de tipos para el sistema, así como el estudio y adaptación de técnicas para representar tipos recursivos y verificar pertenencia a conjuntos coinductivos en presencia de operadores conmutativos, asociativos e idempotentes.

2. PROTOTIPO DE IMPLEMENTACIÓN

Se describe a continuación una primera aproximación al desarrollo de un algoritmo de chequeo de tipos para términos de CAP. Presentamos primero un esquema con el que se aborda el problema de forma general, identificando los sub-problemas que se tratarán al profundizar en la implementación concreta del type checker.

Dados un término anotado s y un contexto de tipado Γ, el trabajo del programa será verificar si existe un juicio de tipado derivable para s y, en caso afirmativo, ofrecer al usuario un μ-type A que satisfaga $\Gamma \vdash s : A$. En rigor, esta presentación no se corresponde con la defición clásica de *type checking* [BDS13], en donde el tipo A es también parte de la entrada. Sin embargo, puede verse que los problemas son equivalentes dado que toda la información de tipo asociada a las variables intervinientes se encuentra anotada en el propio término o en el contexto de tipado. De no ser así, el algoritmo falla, como se verá a continuación. Como contraparte, un algoritmo de inferencia de tipos no recibe estas anotaciones, y al analizar una variable busca asignarle el tipo más general posible. Esto último genera la necesidad de unificar expresiones de tipo a lo largo del proceso de inferencia, lo cual no sucede en *type checking.*

El algoritmo está fuertemente basado en los esquemas de tipado para patrones de la Fig. 1.8 y las variantes dirigidas por sintaxis para tipado de términos de la Fig. 1.9. Consiste, esencialmente, en examinar la estructura de s para decidir qué regla de tipado debe verificarse para obtener un juicio derivable. Es importante aquí contar con esquemas dirigidos por sintaxis: basta con analizar la estructura del término para determinar las condiciones necesarias para tiparlo. En casos donde la validez del juicio dependa a su vez de otros juicios se recurrirá a un llamado recursivo del mismo algoritmo sobre los subtérminos correspondientes.

En la Fig. 2.1 se presenta el pseudocódigo de tc y tcp, las funciones encargadas de chequear el tipo de términos y patrones, respectivamente. Ambas funciones están compuestas por cláusulas excluyentes que aplican a distintas formas de construir expresiones. Cada una se relaciona directamente con una regla de tipado, con excepción de la cláusula de aplicación para términos generales ($\mathsf{tc}(\Gamma, r u)$) que codifica tanto (T-APP-AL) como (T-COMP-AL). Ambas reglas requieren verificar el tipo de los subtérminos r y u, por lo que se resuelven los llamados recursivos para luego decidir en base a los resultados cuál es el esquema a aplicar. Cuando el tipo asignado a r sea un datatype se retorna el tipo $A \mathbin{@} B$ correspondiente a la estructura de datos $r\,u$; en caso contrario se verifican las precondiciones de la regla de aplicación para tipos funcionales. Como indica (T-APP-AL), esto implica obtener un tipo equivalente a A de la forma $\oplus_{i \in 1..n}(A_i \supset B_i)$, fallando en caso contrario. En caso de tratarse A de un tipo de la forma $\mu V.A'$, podría ser necesario también desdoblarlo hasta llegar a una expresión equivalente sin ligadores a la cabeza. La función unfold es la encargada de realizar este trabajo que, como se verá en las secciones siguientes, está muy fuertemente ligado a la representación de los datos que se utilice, por lo que la definición correspondiente se introducirá en el contexto de cada una de las implementaciones concretas presentadas. Una vez conocido el tipo $\oplus_{i \in 1..n}(A_i \supset B_i)$ equivalente a A, se debe verificar que el argumento sea aplicable a cada uno de los tipos $A_i \supset B_i$. Se utiliza aquí la noción de subtipado $\preceq_\mu$, cuya implementación se abordará en las secciones siguientes. Por

$$
\begin{array}{rcl}
\mathsf{tc}(\Gamma, x) & \triangleq & \texttt{if } x \in \Gamma \\
 & & \quad \texttt{then } \Gamma(x) \\
 & & \quad \texttt{else fail} \\
\mathsf{tc}(\Gamma, \mathsf{c}) & \triangleq & \mathbb{c} \\
\mathsf{tc}(\Gamma, (p_i \to_{\theta_i} s_i)_{i \in 1..n}) & \triangleq & \texttt{let } A_i = \mathsf{tcp}(\theta_i, p_i), B_i = \mathsf{tc}(\Gamma \cup \theta_i, s_i) \texttt{ in} \\
 & & \quad \texttt{if } \forall i \in 1..n.\ \forall j \in i+1..n.\ \mathsf{compatible}(p_i : A_i,\ p_j : A_j) \\
 & & \quad\quad \texttt{then } \oplus_{i \in 1..n} A_i \supset \oplus_{i \in 1..n} B_i \\
 & & \quad\quad \texttt{else fail} \\
\mathsf{tc}(\Gamma, r u) & \triangleq & \texttt{let } A = \mathsf{tc}(\Gamma, r), B = \mathsf{tc}(\Gamma, u) \texttt{ in} \\
 & & \quad \texttt{if } A \text{ is a datatype} \\
 & & \quad\quad \texttt{then } A \mathbin{@} B \\
 & & \quad\quad \texttt{else let } \oplus_{i \in 1..n}(A_i \supset B_i) = \mathsf{unfold}(A) \texttt{ in} \\
 & & \quad\quad\quad \texttt{if } \forall i \in 1..n.\mathsf{subtype}(\emptyset, B, A_i) \\
 & & \quad\quad\quad\quad \texttt{then } \oplus_{i \in 1..n} B_i \\
 & & \quad\quad\quad\quad \texttt{else fail}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{tcp}(\theta, x) & \triangleq & \texttt{if } x \in \theta \\
 & & \quad \texttt{then } \theta(x) \\
 & & \quad \texttt{else fail} \\
\mathsf{tcp}(\theta, \mathsf{c}) & \triangleq & \mathbb{c} \\
\mathsf{tcp}(\theta, p\, q) & \triangleq & \texttt{let } A = \mathsf{tcp}(\theta, p), B = \mathsf{tcp}(\theta, q) \texttt{ in} \\
 & & \quad \texttt{if } A \text{ is a datatype} \\
 & & \quad\quad \texttt{then } A \mathbin{@} B \\
 & & \quad\quad \texttt{else fail}
\end{array}
$$

Fig. 2.1: Pseudocódigo del algoritmo de chequeo de tipo para términos y patrones.

último, en los casos en que no esté definido en el contexto Γ un tipo asociado a la variable que se quiere tipar se producirá una falla.

Observar que para tipar abstracciones se verifica, de a pares, la compatibilidad entre las ramas que correspondan. En la Fig.2.2 se muestra el pseudocódigo de esta verificación, definido en los mismos términos que se enunció en la Def. 9.

Como en la definición formal, se utiliza el predicado pcomp para determinar si es necesario imponer la restricción de subtipado entre los tipos asignados a los patrones. Esta verificación se lleva a cabo recorriendo simultáneamente ambos patrones y sus tipos correspondientes. Si ambos patrones son compuestos se asume como precondición que deben tener asignados tipos de la forma $A' \mathbin{@} B'$. Esto ocurre efectivamente aquí, ya que las asignaciones provienen de llamados previos a tcp, que siempre construye tipos de esta forma para patrones compuestos.

Por otra parte, en caso de que alguna de las subexpresiones no sea un patrón compuesto se debe determinar si p subsume a q. Notar que para que esto ocurra p debe ser una variable o bien deben ser ambos la misma constante. De no ser así se habrá encontrado una posición de mismatch. Se analizan, entonces, los símbolos admitidos por ambos tipos en la posición actual (*i.e.* $A\|_\epsilon \cap B\|_\epsilon \neq \varnothing$), observando el constructor a la cabeza de cada tipo y uniendo los

$$\begin{array}{rcl}
\mathsf{compatible}(p : A,\ q : B) & \triangleq & \neg\mathsf{pcomp}(p : A,\ q : B) \vee \mathsf{subtype}(\emptyset, B, A) \\
\mathsf{pcomp}(p : A,\ q : B) & \triangleq & \texttt{if}\ p = p_1\, p_2\ \texttt{and}\ q = q_1\, q_2\ \texttt{then} \\
& & \quad \texttt{let}\ A = A_1 \ @\ A_2,\ B = B_1 \ @\ B_2\ \texttt{in} \\
& & \qquad \mathsf{pcomp}(p_1 : A_1,\ q_1 : B_1)\ \wedge\ \mathsf{pcomp}(p_2 : A_2,\ q_2 : B_2) \\
& & \texttt{else} \\
& & \qquad p = x \vee p = q = \mathsf{c} \vee A\|_\epsilon \cap B\|_\epsilon \neq \varnothing
\end{array}$$

Fig. 2.2: Pseudocódigo del algoritmo de chequeo de compatibilidad para dos patrones.

resultados de las uniones[1]. En este punto se sabe que p es necesariamente un compound o una constante. Esto implica que $A \neq \mu, \oplus$ ya que, como observamos anteriormente, proviene de un llamado previo a tcp. Por lo tanto $A\|_\epsilon = \{\mathbb{c}\}$ o bien $A\|_\epsilon = \{@\}$. Al tener $A\|_\epsilon$ un sólo elemento, $A\|_\epsilon \cap B\|_\epsilon$ puede computarse con costo lineal en el tamaño del segundo tipo.

Alcanza encontrar una posición de mismatch en donde los tipos no admitan símbolos en común para considerar las asignaciones compatibles. De no ser así, el primer patrón podría capturar argumentos aplicables al segundo, y se apela al chequeo de subtipado para determinar la compatibilidad entre las ramas.

Como se mencionó previamente, el esquema general de la Fig. 2.1 no especifica la representación utilizada para las expresiones de términos y tipos de CAP, fundamental para evaluar la complejidad del algoritmo. En particular, la elección de esta representación resulta clave al momento de desdoblar términos (como ocurre al aplicar la función unfold). Por otro lado, tanto el chequeo de compatibilidad como el tratamiento de aplicaciones de una abstracción a su argumento requieren precisar un método para verificar si un tipo es subtipo de otro. Se explica a continuación cómo se modeló el cálculo y se abordaron estos problemas en esta primera etapa.

2.1. Modelado del cálculo

Se decidió adoptar una representación natural de los tipos y términos de CAP, basada en estructuras de datos construidas reflejando fielmente la definición sintáctica de dichas expresiones. Sumado a esto, la naturaleza recursiva de los algoritmos manejados hasta el momento y la comodidad de trabajar en un lenguaje funcional para contrastar la definición de dichos algoritmos con la implementación motivó la elección de *Haskell* para desarrollar el programa. El código fuente correspondiente, adjunto al presente informe, está organizado según la estructura de módulos descrita en la Fig. 2.3.

En la Fig. 2.4 se muestra la definición del tipo algebraico Type utilizado para representar μ-types. Cabe mencionar una diferencia notable entre la gramática de las expresiones de tipos y la representación de dichas expresiones en Haskell. Observar que en la presentación del sistema se distingue el conjunto $\mathcal{T}_D$, incluido en el de los tipos generales $\mathcal{T}$. Si bien en la definición sintáctica se introduce una categoría distinta para expresar los elementos en $\mathcal{T}_D$, utilizar un tipo algebraico separado para estos elementos implicaría que expresiones

[1] En caso de tipos $\mu V.A'$ alcanza con observar el símbolo admitido a la cabeza de A', por lo que no es necesario desdoblar el término.

Módulo	Descripción
Fallible	Funciones auxiliares para manejo de error
Substitution	Tipo abstracto utilizado para representar sustituciones
Term	Representación de términos
Term.Compatibility	Chequeo de compatibilidad
Type	Representación de tipos
Type.Equivalence	Chequeo de equivalencia
Type.MUType	Representación alternativa de tipos utilizando uniones maximales
Type.Subtyping	Chequeo de subtipado
Typechecker	Chequeo de tipos para términos
Util	Funciones auxiliares variadas

Fig. 2.3: Estructura de módulos de la implementación en Haskell

```
data Type  =  TVar Index Sort
           |  TConst Label
           |  TComp Type Type
           |  TFunc  Type Type
           |  TUnion Type Type
           |  TRec Type
```

Fig. 2.4: Representación de μ-types en Haskell.

de la forma $D \oplus D'$ tengan, en Haskell, un tipo distinto al de las uniones generales $A \oplus A$.

Para evitar la duplicación de código asociada a tener dos representaciones para cada construcción, se optó por introducir una componente adicional en las variables, denominada **sort**, que indique si el tipo en cuestión se trata de una variable de datatype o de un tipo general. Se considera que en la implementación integral de un intérprete del cálculo sería responsabilidad del módulo de parsing validar la entrada del usuario y anotar las variables con el sort correcto. Asumiendo, entonces, que el type checker trabaja con expresiones de tipos bien formadas, esta información es suficiente para distinguir datatypes de tipos generales cuando es necesario, realizando una recorrida lineal sobre la expresión.

Con el objetivo de simplificar el trabajo con sustituciones se utilizan **índices de De Bruijn** para representar variables. Cada aparición de una variable se traduce a un número que indica la cantidad de ligadores entre la misma y su ligador correspondiente. De esta forma se elimina la necesidad de introducir nombres en los ligadores y realizar sustituciones innecesarias para evitar capturas de variables en conflicto. Se utilizan, entonces, construcciones de la forma `TRec t` para representar tipos recursivos $\mu V.A$. Por ejemplo, el tipo $\mu X.X \supset \mathbb{c}$ se representa en Haskell como `TRec $ TFunc (TVar 1 T) (TConst "c")`, donde `T` es el constructor que identifica el sort de los tipos generales.

Como se mencionó en la introducción, construcciones de la forma $A \oplus (B \oplus C)$ son equivalentes a cualquier permutación y asociación de sus componentes. Por lo tanto, una vez introducidas por el usuario las anotaciones de tipo respetando la sintaxis de conectores binarios, esta información deja de ser relevante para el programa. Dado que suele ser conveniente, además, trabajar con todos los componentes en simultáneo en vez de considerar

$$
\begin{array}{lll}
\mathsf{unfold}(A) & \triangleq & \texttt{if } A = A' \supset A'' \texttt{ then} \\
& & \quad A \\
& & \texttt{else if } A = \bigoplus_{i \in 1..n} A_i \texttt{ and } n > 1 \texttt{ and } A_i \neq \oplus \texttt{ then} \\
& & \quad \texttt{let } \bigoplus_{j \in 1..m_i} B_{ij} = \mathsf{unfold}(A_i) \texttt{ foreach } i \in 1..n \texttt{ in} \\
& & \qquad \bigoplus_{\substack{i \in 1..n \\ j \in 1..m_i}} B_{ij} \\
& & \texttt{else if } A = \mu V.A' \texttt{ then} \\
& & \quad \mathsf{unfold}(\{\mu V.A/V\}\, A) \\
& & \texttt{else} \\
& & \quad \texttt{fail}
\end{array}
$$

Fig. 2.5: Pseudocódigo de la función unfold.

las uniones de a pares, se introduce una representación adicional para tipos, en donde se detectan y aplanan uniones anidadas en construcciones como $\bigoplus_{i \in 1..n} A_i$ donde $A_i \neq \oplus$. En esta representación, llamada `MUType` (por *maximal union types*), las uniones están compuestas por una lista de tipos, manteniendo el invariante de que los componentes de dichos listados no deben ser a su vez uniones.

Observar que se utilizan términos distintos para representar la expresión $\mu V.A$ y su desdoblamiento $\{\mu V.A/V\}\, A$. Puede ser necesario por lo tanto aplicar explícitamente sustituciones para llevar de una representación a otra, como ocurre por ejemplo en la función unfold, cuyo pseudocódigo se muestra en la Fig. 2.5. Con la representación actual, el algoritmo de unfold consiste en recorrer la estructura del tipo, evaluando en base a su constructor si es posible reescribirlo de la forma deseada. En el caso de encontrarse con un ligador se procede a desdoblar la expresión con el objetivo de analizar el constructor de la expresión resultante. Por ejemplo, al aplicarse sobre $(\mu X.X \supset \mathbb{a} \oplus \mathbb{b}) \oplus (\mathbb{c} \supset \mathbb{d})$ es necesario desdoblar el componente izquierdo de la unión principal para expresar el tipo como $((\mu X.X \supset \mathbb{a} \oplus \mathbb{b}) \supset \mathbb{a} \oplus \mathbb{b}) \oplus (\mathbb{c} \supset \mathbb{d})$.

2.2. Subtipado

La implementación del chequeo de subtipado está basada en la relación dirigida por sintaxis $\preceq_{\vec{\mu}}$, definida como el conjunto coinductivo denotado por las reglas de la Fig. 1.10. Como se indica en Prop. 2, esta relación es equivalente a la noción original de subtipado para μ-types $\preceq_{\mu}$.

El algoritmo utilizado en esta primera etapa es una adaptación del esquema gfp para verificación de pertenencia a conjuntos coinductivos presentado en la introducción (ver Fig. 1.5). Dado un par $\langle A, B\rangle$, la estrategia consiste en obtener el conjunto minimal de pares necesarios para justificar $A \preceq_{\vec{\mu}} B$ y verificar, mediante llamados recursivos, la pertenencia de cada uno a la relación. Observar que para calcular este conjunto de soporte, el esquema se basa en la invertibilidad de la función $\Phi_{\preceq_{\vec{\mu}}}$. Sin embargo esta función no es estrictamente invertible, ya que la regla (S-UNION-R-AL) se solapa con sí misma. Por ejemplo, $\mathbb{c} \preceq_{\vec{\mu}} (\mathbb{c} \oplus \mathbb{d}) \oplus (\mathbb{e} \oplus \mathbb{c})$ pertenece a dos conjuntos $\Phi_{\preceq_{\vec{\mu}}}$-densos:

$$
\begin{array}{lll}
X_1 & = & \{\langle \mathbb{c}, (\mathbb{c} \oplus \mathbb{d}) \oplus (\mathbb{e} \oplus \mathbb{c})\rangle, \langle \mathbb{c}, (\mathbb{c} \oplus \mathbb{d})\rangle, \langle \mathbb{c}, \mathbb{c}\rangle\} \\
X_2 & = & \{\langle \mathbb{c}, (\mathbb{c} \oplus \mathbb{d}) \oplus (\mathbb{e} \oplus \mathbb{c})\rangle, \langle \mathbb{c}, (\mathbb{e} \oplus \mathbb{c})\rangle, \langle \mathbb{c}, \mathbb{c}\rangle\}
\end{array}
$$

```
subtype(S, A, B) \triangleq
    if ⟨A, B⟩ ∈ S then
      S
    else let S_0 = S ∪ {⟨A, B⟩} in
      if A = B = a then
        S_0
      elseif A = A_1 ⊃ A_2 and B = B_1 ⊃ B_2 then
        let S_1 = subtype(S_0, B_1, A_1) in
          subtype(S_1, A_2, B_2)
      elseif A = A_1 @ A_2 and B = B_1 @ B_2 and A_1, B_1 are datatypes then
        let S_1 = subtype(S_0, A_1, B_1) in
          subtype(S_1, A_2, B_2)
      elseif A = μV.A_1 then
        subtype(S_0, {μV.A_1/V} A_1, B)
      elseif A ≠ μ and B = μW.B_1 then
        subtype(S_0, A, {μW.B_1/W} B_1)
      elseif A = \oplus_{i∈1..n} A_i and n > 1 and B ≠ μ and A_i ≠ \oplus then
        let S_1     = subtype(S_0, A_1, B) in
        let S_2     = subtype(S_1, A_2, B) in
        ...
        let S_{n-1} = subtype(S_{n-2}, A_{n-1}, B) in
            subtype(S_{n-1}, A_n, B)
      elseif B = \oplus_{j∈1..m} B_j and m > 1 and A ≠ μ, \oplus and B_j ≠ \oplus then
        seq subtype(S_0, A, B_1), ..., subtype(S_0, A, B_m)
      else
        fail
```

Fig. 2.6: Pseudocódigo del algoritmo de subtipado.

Como se ve a continuación, esto no presenta mayores problemas a la hora de adaptar el esquema gfp a la relación de subtipado. Basta sencillamente con probar cada uno de los componentes de la unión en busca de una verificación exitosa, fallando en caso contrario.

En la Fig. 2.6 se presenta el pseudocódigo del algoritmo. Tal como en el esquema general, se verifica primero si el par de entrada $\langle A, B\rangle$ ya fue verificado y, de no ser así, se chequean los pares necesarios para probar su pertenencia a la relación. Esencialmente, cada condición verificada por el algoritmo se corresponde con una regla que determina los casos recursivos a analizar, representando (S-REFL-AL) un caso base por predicar sobre pares con soporte vacío. La relación entre el pseudocódigo y las reglas es directa excepto por el caso correspondiente a la regla (S-UNION-R-AL), explicado previamente. Se utiliza aquí la función auxiliar `seq`, que evalúa una serie de expresiones retornando el primer resultado exitoso, o produciendo una falla en caso contrario.

2.3. Equivalencia

Modelar μ-types con tipos algebraicos puede dar lugar a la existencia de distintas expresiones que representen el mismo tipo. Por ejemplo, en la implementación actual el tipo $\mathbb{c}\oplus\mathbb{d}$ puede representarse en Haskell como `MUTUnion [MUTConst "c", MUTConst "d"]`

y también como `MUTUnion [MUTConst "d", MUTConst "c"]`. De la misma forma, para representar un tipo recursivo $\mu V.A$ se tendrán distintas expresiones para cualquiera de sus desdoblamientos.

Como se mencionó en la introducción, en sistemas con tipos equi-recursivos se debe permitir al usuario trabajar indistintamente con cualquier expresión concreta de un tipo, siendo responsabilidad del programa el trabajo interno necesario para equiparar estas representaciones cuando sea necesario. Si bien la noción de subtipado basta para desarrollar el chequeo de tipos, se presenta también aquí la implementación de un algoritmo para verificar equivalencia de μ-types, cuyo pseudocódigo se muestra en la Fig. 2.7 . Se utilizó como base también el esquema gfp, esta vez adaptándolo para capturar la relación $\simeq_{\vec{\mu}}$ (ver Fig. 1.11) que, como indica Prop. 3, es equivalente a $\simeq_{\mu}$.

Como indican las reglas (E-REC-L-AL) y (E-REC-R-AL), se deben desdoblar expresiones cuando alguno de los tipos a tratar sea de la forma $\mathscr{C}[\mu V.A]$, donde $\mathscr{C}$ denota contextos

$$A_1 \oplus \ldots A_{i-1} \oplus \Box \oplus A_{i+1} \oplus \ldots \oplus A_n$$

con $A_j \neq \oplus$ para todo $j \in 1..n \setminus i$ y $A_l \neq \mu$ para todo $l \in 1..i-1$. Para identificar estas expresiones y desdoblar los subtérminos correspondientes se utilizan las funciones hasRec y unfoldRec, respectivamente. Dado el invariante de la representación `MUType` mencionado previamente, se puede asumir que los componentes de cada unión no son a su vez uniones, por lo que estas funciones se implementan sencillamente haciendo un recorrido lineal sobre los componentes de la unión maximal y, en el caso de unfoldRec, realizando la sustitución correspondiente.

Por otra parte, en el caso correspondiente a la regla (E-UNION-AL), se debe testear cada componente de uno de los tipos unión involucrados contra todas las componentes del otro, con el objetivo de encontrar su contraparte equivalente. Observar aquí que sólo se reutiliza el conjunto de hipótesis una vez que se ha obtenido una asignación exitosa.

2.4. Análisis de complejidad

Durante la ejecución de tc se realiza un recorrido sobre la estructura completa del término. Nos concentramos en particular en los casos de chequeo de aplicaciones y abstracciones, ya que son aquellos en donde se incurre en verificaciones adicionales que rompen el orden lineal de este recorrido.

En el primer caso, al analizar aplicaciones $r\,u$ se realizan los correspondientes llamados recursivos y, de no ser r una estructura de datos, se utiliza la función unfold para obtener una expresión adecuada para el tipo funcional esperado. Hecho esto, el algoritmo realiza un llamado a subtype por cada tipo de la unión maximal resultante. Por otra parte, para términos de la forma $(p_i \to_{\theta_i} s_i)_{i \in 1..n}$ podría ser necesario verificar la compatibilidad de cada asignación de tipos $p_i : A_i$ con todas las siguientes, lo cual resultaría en $O(n^2)$ llamados a compatible. Esta función auxiliar, por su parte, realiza un recorrido lineal sobre los patrones examinados para evaluar pcomp y eventualmente efectúa un llamado a subtype.

Observar, en primer lugar, que al utilizar la función unfold puede ser necesario desdoblar expresiones de la forma $\mu V.A$ para evaluar si el tipo se trata de una abstracción. Dependiendo de su estructura, el tamaño del resultado de dicha sustitución podría crecer exponencialmente con respecto al de la expresión original. Considerar, por ejemplo, la

$$
\begin{array}{l}
\mathsf{eqType}(S, A, B) \triangleq \\
\quad \texttt{if } \langle A, B\rangle \in S \texttt{ then} \\
\quad\quad S \\
\quad \texttt{else let } S_0 = S \cup \{\langle A, B\rangle\} \texttt{ in} \\
\quad\quad \texttt{if } A = B = a \texttt{ then} \\
\quad\quad\quad S_0 \\
\quad\quad \texttt{elseif } A = A_1 \supset A_2 \texttt{ and } B = B_1 \supset B_2 \texttt{ then} \\
\quad\quad\quad \texttt{let } S_1 = \mathsf{eqType}(S_0, A_1, B_1) \texttt{ in} \\
\quad\quad\quad\quad \mathsf{eqType}(S_1, A_2, B_2) \\
\quad\quad \texttt{elseif } A = A_1 \mathbin{@} A_2 \texttt{ and } B = B_1 \mathbin{@} B_2 \texttt{ and } A_1, B_1 \text{ are datatypes } \texttt{then} \\
\quad\quad\quad \texttt{let } S_1 = \mathsf{eqType}(S_0, A_1, B_1) \texttt{ in} \\
\quad\quad\quad\quad \mathsf{eqType}(S_1, A_2, B_2) \\
\quad\quad \texttt{elseif } \mathsf{hasRec}(A) \texttt{ then} \\
\quad\quad\quad \mathsf{eqType}(S_0, \mathsf{unfoldRec}(A), B) \\
\quad\quad \texttt{elseif not } \mathsf{hasRec}(A) \texttt{ and } \mathsf{hasRec}(B) \texttt{ then} \\
\quad\quad\quad \mathsf{eqType}(S_0, A, \mathsf{unfoldRec}(B)) \\
\quad\quad \texttt{elseif } A = \oplus_{i\in 1..n} A_i \texttt{ and } B = \oplus_{j \in 1..m} B_j \texttt{ and } A_i, B_j \neq \oplus \texttt{ and } n + m > 2 \texttt{ then} \\
\quad\quad\quad \texttt{let } S_1 = (\mathsf{seq}\ \mathsf{eqType}(S_0, A_1, B_1), \ldots, \mathsf{eqType}(S_0, A_1, B_m)) \texttt{ in} \\
\quad\quad\quad \ldots \\
\quad\quad\quad \texttt{let } S_n = (\mathsf{seq}\ \mathsf{eqType}(S_{n-1}, A_n, B_1), \ldots, \mathsf{eqType}(S_{n-1}, A_n, B_m)) \texttt{ in} \\
\quad\quad\quad \texttt{let } S_{n+1} = (\mathsf{seq}\ \mathsf{eqType}(S_n, A_1, B_1), \ldots, \mathsf{eqType}(S_n, A_n, B_1)) \texttt{ in} \\
\quad\quad\quad \ldots \\
\quad\quad\quad \texttt{let } S_{n+m-1} = (\mathsf{seq}\ \mathsf{eqType}(S_{n+m-2}, A_1, B_{m-1}), \ldots, \mathsf{eqType}(S_{n+m-2}, A_n, B_{m-1})) \texttt{ in} \\
\quad\quad\quad\quad \mathsf{seq}\ \mathsf{eqType}(S_{n+m-1}, A_1, B_m), \ldots, \mathsf{eqType}(S_{n+m-1}, A_n, B_m) \\
\quad\quad \texttt{else} \\
\quad\quad\quad \texttt{fail}
\end{array}
$$

Fig. 2.7: Pseudocódigo del algoritmo de equivalencia.

secuencia de desdoblamientos del tipo $\mu X_1 . \ldots \mu X_n . \oplus_{i \in 1..n}(X_i \supset \mathbb{c})$:

$$\begin{array}{rcl} t_0 & = & \mu X_1 . \ldots \mu X_n . \oplus_{i \in 1..n}(X_i \supset \mathbb{c}) \\ t_1 & = & \mu X_2 . \ldots \mu X_n . \oplus_{i \in 2..n}(X_i \supset \mathbb{c}) \oplus (t_0 \supset \mathbb{c}) \\ t_2 & = & \mu X_3 . \ldots \mu X_n . \oplus_{i \in 3..n}(X_i \supset \mathbb{c}) \oplus (t_0 \supset \mathbb{c}) \oplus (t_1 \supset \mathbb{c}) \\ & \ldots & \\ t_n & = & \oplus_{i \in 1..n}(t_i \supset \mathbb{c}) \end{array}$$

Aquí, en cada reescritura se remplaza una variable por la expresión entera, duplicando el tamaño de la misma en cada paso, obteniendo un resultado final con tamaño de orden 2^n.

Un problema similar surge en la función `subtype`, donde se deben realizar las sustituciones adecuadas para desdoblar construcciones recursivas. Adicionalmente, en [Pie02] se argumenta que el esquema `gfp` utilizado para verificar subtipado produce una cantidad cuadrática de llamados recursivos, producto de explorar las combinaciones de los subtérminos del par a verificar. Esta justificación está basada fuertemente en el hecho de que nunca se descarta el conjunto de hipótesis confirmadas hasta el momento. Sin embargo, este no es el caso de nuestras adaptaciones. Al verificar tipos de la forma $A \preceq_\mu \oplus_{i \in 1..m} B_j$ (con $m > 1, A \neq \mu, \oplus$ and $B_j \neq \oplus$) pueden llegar a efectuarse m llamados recursivos realizando un *backtracking* en cada paso, producto de descartar el conjunto de hipótesis tras cada prueba de la función `seq`. Sumado a esto, el hecho de que la cantidad de subproblemas dependa de la estructura de los tipos involucrados dificulta un análisis detallado sobre la cantidad total de operaciones en un algoritmo de naturaleza recursiva como el que presentamos. No conocemos, en particular, herramientas que permitan realizar este tipo de análisis en presencia de operadores asociativos, conmutativos e idempotentes.

2.5. Discusión

La principal ventaja de esta primera implementación del algoritmo de type checking es su claridad. Por un lado, dada la definición formal de la sintaxis del lenguaje, resulta natural la utilización de tipos algebraicos para representar y manipular las expresiones de tipos. Por otra parte, contar con un marco general como el esquema `gfp` es conveniente para razonar sobre el funcionamiento del chequeo de subtipado y equivalencia. La estructura del algoritmo derivado presenta, además, una relación muy estrecha con las reglas que definen la relación. A pesar de esto, como se mostró, las limitaciones de este enfoque se evidencian a la hora de hacer un análisis fino sobre la complejidad del algoritmo.

Por otro lado, la elección de un lenguaje de programación funcional puro como Haskell resulta ventajosa a la hora de contrastar la definición teórica del algoritmo con su implementación. No obstante, desde un punto de vista práctico, el mecanismo de evaluación *lazy* del lenguaje introduce una dimensión adicional al estudio del comportamiento del algoritmo en tiempo de ejecución.

3. IMPLEMENTACIÓN EFICIENTE

Describimos de aquí en adelante una segunda implementación desarrollada para lidiar con los problemas mencionados anteriormente. El foco en esta etapa estuvo en permitir un análisis certero sobre la complejidad del algoritmo de chequeo de tipos y, más allá del comportamiento asintótico, obtener una implementación robusta y eficiente en la práctica.

Como se verá a continuación, los cambios fundamentales con respecto a la versión anterior tienen que ver con la adopción de una nueva representación para las expresiones de tipos (basada en estructuras enlazadas potencialmente cíclicas) y la introducción de un nuevo algoritmo para la verificación de subtipado de espíritu más secuencial que el basado en el esquema gfp. Con el objetivo de abordar estos subproblemas de forma imperativa se decidió llevar a cabo el desarrollo en *Scala*, lenguaje que permitió a su vez mantener en líneas generales la declaratividad y ciertos aspectos funcionales de la implementación anterior. Para facilitar la lectura, se describe en la Fig. 3.1 la estructura del código fuente correspondiente.

3.1. Cambio de representación

Como se mencionó en la sección anterior, la primera implementación del type checker utiliza internamente representaciones distintas para expresiones de la forma $\mu V.A$ y su desdoblamiento $\{\mu V.A/V\}\,A$. La consecuencia inmediata de esto es que a la hora de analizar la estructura del tipo $\mu V.A$ puede ser necesario computar la sustitución necesaria para obtener una expresión alternativa del mismo. Dado que el resultado de aplicar la sustitución puede crecer exponencialmente, el costo de desdoblar un término termina gobernando la complejidad total del algoritmo, tornándolo inaplicable a efectos prácticos.

Antes de describir la solución al problema se presenta una nueva noción de tipo, sobre la cual se construye la nueva representación utilizada. Partimos del alfabeto de símbolos con aridad $\mathfrak{L}^{\mathfrak{n}} \triangleq \left\{a^0 \mid a \in \mathcal{V} \cup \mathcal{C}\right\} \cup \left\{@^2, \supset^2\right\} \cup \{\oplus^n \mid n > 1\}$ y definimos $\mathfrak{T}^{\mathfrak{n}}$ como el conjunto de árboles posiblemente infinitos con símbolos en $\mathfrak{L}^{\mathfrak{n}}$.

Definición 11. *Un árbol (posiblemente) infinito con uniones n-arias $\mathcal{A} \in \mathfrak{T}^{\mathfrak{n}}$ es una función parcial de posiciones a símbolos en $\mathfrak{L}^{\mathfrak{n}}$ tal que:*

- *el dominio de $\mathcal{A}$ es no vacío y cerrado por prefijos*
- *si $\mathcal{A}(\pi) = \star$ entonces $\{i \mid \pi i \in \mathsf{dom}\,(\mathcal{A})\} = 1..k$, siendo k la aridad de $\star$*

Cuando quede claro por el contexto que nos referimos a árboles en $\mathfrak{T}^{\mathfrak{n}}$, utilizaremos para facilitar la lectura la siguiente notación. Si $\star \in \left\{@^2, \supset^2\right\}$, denotamos con $\mathcal{B} \star \mathcal{C}$ el árbol $\mathcal{A}$ tal que $\mathcal{A}(\epsilon) = \star$ y para toda posición π vale $\mathcal{A}(1\pi) = \mathcal{B}(\pi)$ y $\mathcal{A}(2\pi) = \mathcal{C}(\pi)$. De forma similar, denotamos con $\oplus_i^n \mathcal{A}_i$ el árbol $\mathcal{A}$ tal que $\mathcal{A}(\epsilon) = \oplus^n$ y $\mathcal{A}(i\pi) = \mathcal{A}_i(\pi)$.

Para relacionar las nuevas expresiones de tipos con el sistema original se utiliza una traducción intermedia a $\mathfrak{T}$, el conjunto de árboles binarios presentado en la introducción. La traducción se comporta de la forma esperada, introduciendo uniones n-arias para reemplazar uniones binarias maximales en elementos de $\mathfrak{T}$.

Módulo/clase	Descripción
compatibility	Chequeo de compatibilidad
conversions/nodeToType	Conversión de representación `TypeNode` a `MUType`
conversions/typeToNode	Conversión de representación `MUType` a `TypeNode`
DOTBuilder	Generación de representaciones visuales de los *term automata*
dsl	Definiciones auxiliares para facilitar la escritura de expresiones
equivalence	Chequeo de equivalencia
gfp	Esquema general para computar $\preceq_\mu$ y $\simeq_\mu$
Graph	Representación genérica de grafos dirigidos
MUType	Representación de tipos utilizando uniones maximales
package	Definiciones auxiliares globales
subtyping	Chequeo de subtipado
Term	Representación de términos
Type	Representación de tipos con uniones binarias
typechecking	Chequeo de tipos para términos
TypecheckingError	Excepciones utilizadas para manejo de error del type checker
TypeNode	Representación de tipos como *term automata*

Fig. 3.1: Estructura de módulos de la implementación en Scala

Definición 12. *La traducción* $[\![\cdot]\!]^{\mathsf{n}} : \mathfrak{T} \to \mathfrak{T}^{\mathsf{n}}$ *se define por inducción en las posiciones, del siguiente modo:*

$$\begin{array}{rcll} [\![a]\!]^{\mathsf{n}}(\epsilon) & \triangleq & a & \\ [\![\mathcal{A}_1 \star \mathcal{A}_2]\!]^{\mathsf{n}}(\epsilon) & \triangleq & \star & \star \in \{@, \supset\} \\ [\![\mathcal{A}_1 \star \mathcal{A}_2]\!]^{\mathsf{n}}(i\pi) & \triangleq & [\![\mathcal{A}_i]\!]^{\mathsf{n}}(\pi) & \star \in \{@, \supset\},\ i \in \{1, 2\} \\ [\![\oplus_{i \in 1..n}\mathcal{A}_i]\!]^{\mathsf{n}}(\epsilon) & \triangleq & \oplus^n & \mathcal{A}_i \neq \oplus \\ [\![\oplus_{i \in 1..n}\mathcal{A}_i]\!]^{\mathsf{n}}(i\pi) & \triangleq & [\![\mathcal{A}_i]\!]^{\mathsf{n}}(\pi) & \mathcal{A}_i \neq \oplus \end{array}$$

Considerar por ejemplo $\mathsf{List}_A = \mu\alpha.\mathbb{nil} \oplus (\mathbb{cons} \; @ \; A \; @ \; \alpha)$, el μ-type de las listas con elementos de tipo A, cuya interpretación como árbol infinito se ilustra en la Fig. 3.2. La técnica utilizada en la implementación para representar concretamente los árboles infinitos, presentada en [AC93], está basada en la interpretación de estos árboles como *term automata*.

Definición 13. *Un **term automaton** es una tupla*

$$\mathcal{M} \;\; = \;\; (Q,\ \Sigma,\ q_0,\ \delta,\ \ell)$$

donde:

1. Q *es un conjunto finito de estados*
2. Σ *es un alfabeto donde cada símbolo tiene asociada una aridad*
3. q_0 *es el estado inicial*
4. $\delta : Q \times \mathbb{N} \to Q$ *es una función parcial de transición entre estados, definida para el rango* $1..k$, *donde* k *es la aridad del símbolo asociado por* ℓ *al estado de partida*

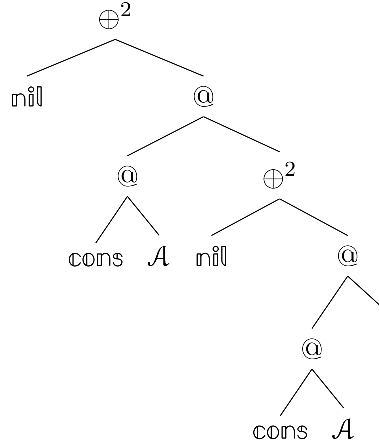


Fig. 3.2: Interpretación de List_A como un árbol infinito, donde $\mathcal{A}$ denota $[\![[\![A]\!]^{\mathfrak{T}}]\!]^{\mathfrak{n}}$.

5. *$\ell : Q \to \Sigma$ es una función (total) que asocia una etiqueta del alfabeto Σ a cada estado.*

Denotamos con $\mathcal{M}_A$ al autómata asociado al tipo A. Intuitivamente, un autómata permite reconocer las cadenas de transiciones asociadas a los caminos del árbol comenzando desde la raíz. Retomando el ejemplo anterior, si llamamos q_0 al estado inicial de $\mathcal{M}_{\mathsf{List}_A}$ y consideramos la extensión natural de la función de transición a cadenas $\widehat{\delta}$, entonces $\ell(\widehat{\delta}(q_0, 211)) = \mathbb{cons}$. La estructura regular de los árboles nos permite utilizar la representación clásica de autómata como grafos cíclicos. Como se ve en la Fig. 3.3, esta representación compacta el árbol en una estructura finita y captura cualquier desdoblamiento del μ-type asociado.

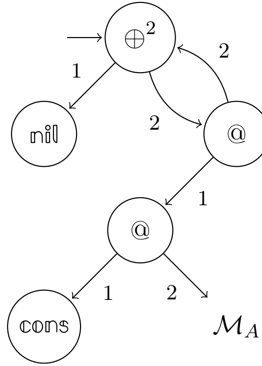


Fig. 3.3: Representación de $\mathcal{M}_{\mathsf{List}_A}$ como un grafo cíclico.

En la implementación, dado un tipo $\mu V.A$, se construye explícitamente el grafo como una estructura enlazada en donde cualquier aparición de la variable V dentro de A será representada mediante un puntero al nodo de inicio. Al desaparecer los ligadores, esta representación elimina por completo el problema de desdoblar una expresión para inspeccionar su estructura. Observar que, al introducir este cambio, la función unfold utilizada durante el algoritmo de subtipado se debe limitar a una verificación lineal sobre su argumento, produciendo una falla en caso de que no sea un tipo funcional (*i.e.* una función o una unión de funciones).

$$\overline{\overline{a \preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} a}} \ (\text{S-REFL-UP})$$

$$\frac{\mathcal{D} \ (\preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \cup \mathcal{R}) \ \mathcal{D}' \quad \mathcal{A} \ (\preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \cup \mathcal{R}) \ \mathcal{A}'}{\mathcal{D} \ @ \ \mathcal{A} \preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \mathcal{D}' \ @ \ \mathcal{A}'} \ (\text{S-COMP-UP}) \qquad \frac{\mathcal{A}' \ (\preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \cup \mathcal{R}) \ \mathcal{A} \quad \mathcal{B} \ (\preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \cup \mathcal{R}) \ \mathcal{B}'}{\mathcal{A} \supset \mathcal{B} \preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \mathcal{A}' \supset \mathcal{B}'} \ (\text{S-FUNC-UP})$$

$$\frac{\mathcal{A}_i \ (\preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \cup \mathcal{R}) \ \mathcal{B}_{f(i)} \quad f : 1..n \to 1..m \quad \mathcal{A}_i, \mathcal{B}_j \neq \oplus}{\bigoplus_i^n \mathcal{A}_i \preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \bigoplus_j^m \mathcal{B}_j} \ (\text{S-UNION-UP})$$

$$\frac{\mathcal{A}_i \ (\preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \cup \mathcal{R}) \ \mathcal{B} \text{ para todo } i \in 1..n \quad \mathcal{A}_i \neq \oplus \quad \mathcal{B} \neq \oplus}{\bigoplus_i^n \mathcal{A}_i \preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \mathcal{B}} \ (\text{S-UNION-L-UP})$$

$$\frac{\mathcal{A} \ (\preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \cup \mathcal{R}) \ \mathcal{B}_k \text{ para algún } k \in 1..m \quad \mathcal{A} \neq \oplus \quad \mathcal{B}_j \neq \oplus}{\mathcal{A} \preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \bigoplus_j^m \mathcal{B}_j} \ (\text{S-UNION-R-UP})$$

Fig. 3.4: Relación de subtipado para árboles infinitos con uniones n-arias

3.2. Subtipado

Se presenta a continuación la relación $\preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}}$ de subtipado *up-to* un conjunto de hipótesis $\mathcal{R}$ (ver Fig. 3.4). Luego, formalizamos la noción de subtipado $\preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}$ para $\mathfrak{T}^{\mathfrak{n}}$ como un caso particular, y probamos su correspondencia con $\preceq_\mu$ vía una traducción apropiada entre sus dominios, para garantizar que puede computarse esta última utilizando la nueva formulación.

Definición 14. *Se define $\preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}}$ como $\nu\Phi_{\preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}}}$, donde $\Phi_{\preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}}} : \wp(\mathfrak{T}^{\mathfrak{n}} \times \mathfrak{T}^{\mathfrak{n}}) \to \wp(\mathfrak{T}^{\mathfrak{n}} \times \mathfrak{T}^{\mathfrak{n}})$ es la siguiente función:*

$$\begin{array}{rcl}
\Phi_{\preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}}}(\mathcal{S}) & = & \{\langle a, a\rangle \mid a \in \mathcal{V} \cup \mathcal{C}\} \\
& \cup & \{\langle \mathcal{D} \ @ \ \mathcal{A}, \mathcal{D}' \ @ \ \mathcal{A}'\rangle \mid \langle \mathcal{D}, \mathcal{D}'\rangle, \langle \mathcal{A}, \mathcal{A}'\rangle \in (\mathcal{S} \cup \mathcal{R})\} \\
& \cup & \{\langle \mathcal{A} \supset \mathcal{B}, \mathcal{A}' \supset \mathcal{B}'\rangle \mid \langle \mathcal{A}', \mathcal{A}\rangle, \langle \mathcal{B}, \mathcal{B}'\rangle \in (\mathcal{S} \cup \mathcal{R})\} \\
& \cup & \{\langle \oplus_i^n \mathcal{A}_i, \oplus_j^m \mathcal{B}_j\rangle \mid \mathcal{A}_i, \mathcal{B}_j \neq \oplus, \\
& & \quad \exists f : 1..n \to 1..m \textit{ tal que } \langle \mathcal{A}_i, \mathcal{B}_{f(i)}\rangle \in (\mathcal{S} \cup \mathcal{R})\} \\
& \cup & \{\langle \oplus_i^n \mathcal{A}_i, \mathcal{B}\rangle \mid \mathcal{A}_i, \mathcal{B} \neq \oplus, \forall i \in 1..n. \langle \mathcal{A}_i, \mathcal{B}\rangle \in (\mathcal{S} \cup \mathcal{R})\} \\
& \cup & \{\langle \mathcal{A}, \oplus_j^m \mathcal{B}_j\rangle \mid \mathcal{A}, \mathcal{B}_j \neq \oplus, \exists k \in 1..m. \langle \mathcal{A}, \mathcal{B}_k\rangle \in (\mathcal{S} \cup \mathcal{R})\}
\end{array}$$

Lema 3. *Para todo $\mathcal{A}, \mathcal{B} \in \mathfrak{T}$, $\mathcal{A} \preceq_{\mathfrak{T}} \mathcal{B}$ si y sólo si $[\![\mathcal{A}]\!]^{\mathfrak{n}} \preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}]\!]^{\mathfrak{n}}$.*

Demostración. $\Rightarrow$) Probamos esta implicación mostrando que $\mathcal{R} \triangleq \{\langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}}\rangle \mid \mathcal{A} \preceq_{\mathfrak{T}} \mathcal{B}\}$ es $\Phi_{\preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}}$-denso. Se procede analizando la estructura de cualquier elemento posible de $\mathcal{R}$.

- $\langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}}\rangle = \langle a, a\rangle$. Entonces $\langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}}\rangle \in \Phi_{\preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}}(\mathcal{R})$.
- $\langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}}\rangle = \langle \mathcal{T}' \ @ \ \mathcal{T}'', \mathcal{S}' \ @ \ \mathcal{S}''\rangle$. Notar que la traducción de un tipo en $\mathfrak{T}$ sólo puede tener un símbolo @ en la raíz si el tipo original también lo tiene. De aquí se infiere

que $\mathcal{A}$ debe ser de la forma $\mathcal{D}$ @ $\mathcal{A}'$, donde $[\![\mathcal{D}]\!]^{\mathfrak{n}} = \mathcal{T}'$ y $[\![\mathcal{A}']\!]^{\mathfrak{n}} = \mathcal{T}''$. De la misma manera sabemos que $\mathcal{B}$ es de la forma $\mathcal{D}'$ @ $\mathcal{B}'$, con $[\![\mathcal{D}']\!]^{\mathfrak{n}} = \mathcal{S}'$ y $[\![\mathcal{B}']\!]^{\mathfrak{n}} = \mathcal{S}''$.

Dado que $\mathcal{A} \preceq_{\mathfrak{T}} \mathcal{B}$, sabemos que valen tanto $\mathcal{D} \preceq_{\mathfrak{T}} \mathcal{D}'$ como $\mathcal{A}' \preceq_{\mathfrak{T}} \mathcal{B}'$, lo que significa que $\{\langle [\![\mathcal{D}]\!]^{\mathfrak{n}}, [\![\mathcal{D}']\!]^{\mathfrak{n}}\rangle, \langle [\![\mathcal{A}']\!]^{\mathfrak{n}}, [\![\mathcal{B}']\!]^{\mathfrak{n}}\rangle\} \subseteq \mathcal{R}$. Por lo tanto, $\langle [\![\mathcal{D}]\!]^{\mathfrak{n}}$ @ $[\![\mathcal{A}']\!]^{\mathfrak{n}}, [\![\mathcal{D}']\!]^{\mathfrak{n}}$ @ $[\![\mathcal{B}']\!]^{\mathfrak{n}}\rangle \in \Phi_{\preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}}(\mathcal{R})$. Usando nuevamente la definición de $[\![\cdot]\!]^{\mathfrak{n}}$, tenemos

$$\begin{array}{rcl} \langle [\![\mathcal{D}]\!]^{\mathfrak{n}} \mathbin{@} [\![\mathcal{A}']\!]^{\mathfrak{n}}, [\![\mathcal{D}']\!]^{\mathfrak{n}} \mathbin{@} [\![\mathcal{B}']\!]^{\mathfrak{n}}\rangle & = & \langle [\![\mathcal{D} \mathbin{@} \mathcal{A}']\!]^{\mathfrak{n}}, [\![\mathcal{D}' \mathbin{@} \mathcal{B}']\!]^{\mathfrak{n}}\rangle \\ & = & \langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}}\rangle \in \Phi_{\preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}}(\mathcal{R}) \end{array}$$

- $\langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}}\rangle = \langle \mathcal{T}' \supset \mathcal{T}'', \mathcal{S}' \supset \mathcal{S}''\rangle$. Este caso es similar al anterior. Primero inferimos que $\langle \mathcal{A}, \mathcal{B}\rangle$ es de la forma $\langle \mathcal{A}' \supset \mathcal{A}'', \mathcal{B}' \supset \mathcal{B}''\rangle$, donde $[\![\mathcal{A}']\!]^{\mathfrak{n}} = \mathcal{T}'$, $[\![\mathcal{A}'']\!]^{\mathfrak{n}} = \mathcal{T}''$, $[\![\mathcal{B}']\!]^{\mathfrak{n}} = \mathcal{S}'$, $[\![\mathcal{B}'']\!]^{\mathfrak{n}} = \mathcal{S}''$. De $\mathcal{A} \preceq_{\mathfrak{T}} \mathcal{B}$ obtenemos $\{\langle [\![\mathcal{B}']\!]^{\mathfrak{n}}, [\![\mathcal{A}']\!]^{\mathfrak{n}}\rangle, \langle [\![\mathcal{A}'']\!]^{\mathfrak{n}}, [\![\mathcal{B}'']\!]^{\mathfrak{n}}\rangle\} \subseteq \mathcal{R}$, lo que significa que

$$\begin{array}{rcl} \langle [\![\mathcal{A}']\!]^{\mathfrak{n}} \supset [\![\mathcal{A}'']\!]^{\mathfrak{n}}, [\![\mathcal{B}']\!]^{\mathfrak{n}} \supset [\![\mathcal{B}'']\!]^{\mathfrak{n}}\rangle & = & \langle [\![\mathcal{A}' \supset \mathcal{A}'']\!]^{\mathfrak{n}}, [\![\mathcal{B}' \supset \mathcal{B}'']\!]^{\mathfrak{n}}\rangle \\ & = & \langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}}\rangle \in \Phi_{\preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}}(\mathcal{R}) \end{array}$$

- $\langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}}\rangle = \langle \oplus_i^n \mathcal{T}_i, \mathcal{S}\rangle$ con $\mathcal{S} \neq \oplus$. Un árbol de la forma $\oplus_i^n \mathcal{T}_i$ sólo puede ser resultado de traducir una unión maximal de $n > 1$ elementos, por lo que $\mathcal{A}$ debe ser $\oplus_{i\in 1..n}\mathcal{A}_i$, donde $[\![\mathcal{A}_i]\!]^{\mathfrak{n}} = \mathcal{T}_i$. Al mismo tiempo $\mathcal{S} \neq \oplus$ implica $\mathcal{B} \neq \oplus$, por lo que de $\mathcal{A} \preceq_{\mathfrak{T}} \mathcal{B}$ sabemos que para todo $i \in 1..n$ vale $\mathcal{A}_i \preceq_{\mathfrak{T}} \mathcal{B}$. Luego, $\langle [\![\mathcal{A}_i]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}}\rangle \in \mathcal{R}$ para todo $i \in 1..n$. Usando la definición de $\Phi_{\preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}}$ y la traducción obtenemos

$$\begin{array}{rcl} \langle \oplus_i^n [\![\mathcal{A}_i]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}}\rangle & = & \langle [\![\oplus_{i\in 1..n}\mathcal{A}_i]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}}\rangle \\ & = & \langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}}\rangle \in \Phi_{\preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}}(\mathcal{R}) \end{array}$$

- $\langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}}\rangle = \langle \mathcal{T}, \oplus_j^m \mathcal{S}_j\rangle$ con $\mathcal{T} \neq \oplus$. De forma similar al caso anterior, podemos asegurar que $\mathcal{A} \neq \oplus$ y $\mathcal{B} = \oplus_{j\in 1..m}\mathcal{B}_j$, donde $m > 1$ y $[\![\mathcal{B}_j]\!]^{\mathfrak{n}} = \mathcal{S}_j$. Dado que $\mathcal{A} \preceq_{\mathfrak{T}} \mathcal{B}$, existe un $j \in 1..m$ tal que $\mathcal{A} \preceq_{\mathfrak{T}} \mathcal{B}_j$ y por ende $\langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\mathcal{B}_j]\!]^{\mathfrak{n}}\rangle \in \mathcal{R}$. Sabemos entonces que

$$\begin{array}{rcl} \langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, \oplus_j^m [\![\mathcal{B}_j]\!]^{\mathfrak{n}}\rangle & = & \langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\oplus_{j\in 1..m}\mathcal{B}_j]\!]^{\mathfrak{n}}\rangle \\ & = & \langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}}\rangle \in \Phi_{\preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}}(\mathcal{R}) \end{array}$$

- $\langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}}\rangle = \langle \oplus_i^n \mathcal{T}_i, \oplus_j^m \mathcal{S}_j\rangle$. Usamos el mismo argumento de los dos casos anteriores para inferir que se tienen uniones maximales $\mathcal{A} = \oplus_{i\in 1..n}\mathcal{A}_i$ y $\mathcal{B} = \oplus_{j\in 1..m}\mathcal{B}_j$, donde $[\![\mathcal{A}_i]\!]^{\mathfrak{n}} = \mathcal{T}_i$ y $[\![\mathcal{B}_j]\!]^{\mathfrak{n}} = \mathcal{S}_j$. De $\mathcal{A} \preceq_{\mathfrak{T}} \mathcal{B}$ sabemos que existe $f : 1..n \to 1..m$ tal que $\mathcal{A}_i \preceq_{\mathfrak{T}} \mathcal{B}_{f(i)}$ para todo $i \in 1..n$, por lo que $\langle [\![\mathcal{A}_i]\!]^{\mathfrak{n}}, [\![\mathcal{B}_{f(i)}]\!]^{\mathfrak{n}}\rangle \in \mathcal{R}$. De nuevo, por definición de $\Phi_{\preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}}$ y de la traducción obtenemos

$$\begin{array}{rcl} \langle \oplus_i^n [\![\mathcal{A}_i]\!]^{\mathfrak{n}}, \oplus_j^m [\![\mathcal{B}_j]\!]^{\mathfrak{n}}\rangle & = & \langle [\![\oplus_{i\in 1..n}\mathcal{A}_i]\!]^{\mathfrak{n}}, [\![\oplus_{j\in 1..m}\mathcal{B}_j]\!]^{\mathfrak{n}}\rangle \\ & = & \langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}}\rangle \in \Phi_{\preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}}(\mathcal{R}) \end{array}$$

$\Leftarrow$) De modo similar, definimos $\mathcal{R} = \left\{\langle \mathcal{A}, \mathcal{B}\rangle \mid [\![\mathcal{A}]\!]^{\mathfrak{n}} \preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}]\!]^{\mathfrak{n}}\right\}$ y mostramos que es $\Phi_{\preceq_{\mathfrak{T}}}$-denso. Procedemos analizando la forma de cualquier posible par $\langle \mathcal{A}, \mathcal{B}\rangle \in R$:

- $\langle \mathcal{A}, \mathcal{B}\rangle = \langle a, a\rangle$. Entonces $\langle \mathcal{A}, \mathcal{B}\rangle \in \mathcal{R}$.

- $\langle \mathcal{A}, \mathcal{B}\rangle = \langle \mathcal{D} \mathbin{@} \mathcal{A}', \mathcal{D}' \mathbin{@} \mathcal{B}'\rangle$. La definición de $[\![\cdot]\!]^{\mathfrak{n}}$ nos dice que $\langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}}\rangle$ se define como $\langle [\![\mathcal{D}]\!]^{\mathfrak{n}} \mathbin{@} [\![\mathcal{A}']\!]^{\mathfrak{n}}, [\![\mathcal{D}']\!]^{\mathfrak{n}} \mathbin{@} [\![\mathcal{B}']\!]^{\mathfrak{n}}\rangle$. Además, dado $[\![\mathcal{A}]\!]^{\mathfrak{n}} \preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}]\!]^{\mathfrak{n}}$, deben valer tanto $[\![\mathcal{D}]\!]^{\mathfrak{n}} \preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{D}']\!]^{\mathfrak{n}}$ como $[\![\mathcal{A}']\!]^{\mathfrak{n}} \preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}']\!]^{\mathfrak{n}}$. Esto significa que $\{\langle \mathcal{D}, \mathcal{D}'\rangle, \langle \mathcal{A}', \mathcal{B}'\rangle\} \subseteq \mathcal{R}$, lo que implica
$$\langle \mathcal{A}, \mathcal{B}\rangle \;=\; \langle \mathcal{D}' \mathbin{@} \mathcal{A}', \mathcal{D}' \mathbin{@} \mathcal{B}'\rangle \in \Phi_{\preceq_{\mathfrak{T}}}(\mathcal{R})$$

- $\langle \mathcal{A}, \mathcal{B}\rangle = \langle \mathcal{A}' \supset \mathcal{A}'', \mathcal{B}' \supset \mathcal{B}''\rangle$. Usando un razonamiento similar al anterior, $\langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}}\rangle = \langle [\![\mathcal{A}']\!]^{\mathfrak{n}} \supset [\![\mathcal{A}'']\!]^{\mathfrak{n}}, [\![\mathcal{B}']\!]^{\mathfrak{n}} \supset [\![\mathcal{B}'']\!]^{\mathfrak{n}}\rangle$. Por otro lado, de $[\![\mathcal{A}]\!]^{\mathfrak{n}} \preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}]\!]^{\mathfrak{n}}$ obtenemos $[\![\mathcal{B}']\!]^{\mathfrak{n}} \preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{A}']\!]^{\mathfrak{n}}$ y $[\![\mathcal{A}'']\!]^{\mathfrak{n}} \preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}'']\!]^{\mathfrak{n}}$, por lo que $\{\langle \mathcal{B}', \mathcal{A}'\rangle, \langle \mathcal{A}'', \mathcal{B}''\rangle\} \subseteq \mathcal{R}$. Entonces,
$$\langle \mathcal{A}, \mathcal{B}\rangle \;=\; \langle \mathcal{A}' \supset \mathcal{A}'', \mathcal{B}' \supset \mathcal{B}''\rangle \in \Phi_{\preceq_{\mathfrak{T}}}(\mathcal{R})$$

- $\langle \mathcal{A}, \mathcal{B}\rangle = \langle \oplus_{i\in 1..n}\mathcal{A}_i, \oplus_{j\in 1..m}\mathcal{B}_j\rangle$, con $n + m > 2$ y $\mathcal{A}_i, \mathcal{B}_i \neq \oplus$.

  - Si $m = 1$ entonces $n > 1$, por lo que $[\![\mathcal{A}]\!]^{\mathfrak{n}} = \oplus_i^n [\![\mathcal{A}_i]\!]^{\mathfrak{n}}$. Dado que $[\![\mathcal{A}]\!]^{\mathfrak{n}} \preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}]\!]^{\mathfrak{n}}$, para todo $i \in 1..n$ tenemos $[\![\mathcal{A}_i]\!]^{\mathfrak{n}} \preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}]\!]^{\mathfrak{n}}$ y también $\langle \mathcal{A}_i, \mathcal{B}\rangle \in \mathcal{R}$. La definición de $\Phi_{\preceq_{\mathfrak{T}}}$ nos dice que
  $$\langle \mathcal{A}, \mathcal{B}\rangle \;=\; \langle \oplus_{i\in 1..n}\mathcal{A}_i, \mathcal{B}\rangle \in \Phi_{\preceq_{\mathfrak{T}}}(\mathcal{R})$$

  - Si $n = 1$ entonces $m > 1$, por lo que $[\![\mathcal{B}]\!]^{\mathfrak{n}} = \oplus_j^m [\![\mathcal{B}_j]\!]^{\mathfrak{n}}$. $[\![\mathcal{A}]\!]^{\mathfrak{n}} \preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}]\!]^{\mathfrak{n}}$ significa que existe $j \in 1..m$ tal que $[\![\mathcal{A}]\!]^{\mathfrak{n}} \preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}_j]\!]^{\mathfrak{n}}$ y consecuentemente $\langle \mathcal{A}, \mathcal{B}_j\rangle \in \mathcal{R}$. Por lo tanto,
  $$\langle \mathcal{A}, \mathcal{B}\rangle \;=\; \langle \mathcal{A}, \oplus_{j\in 1..m}\mathcal{B}_j\rangle \in \Phi_{\preceq_{\mathfrak{T}}}(\mathcal{R})$$

  - Si $n > 1$ y $m > 1$ tanto $\mathcal{A}$ como $\mathcal{B}$ son uniones maximales de al menos dos elementos. Como en los casos anteriores, inferimos que $[\![\mathcal{A}]\!]^{\mathfrak{n}} = \oplus_i^n [\![\mathcal{A}_i]\!]^{\mathfrak{n}}$, $[\![\mathcal{B}]\!]^{\mathfrak{n}} = \oplus_j^m [\![\mathcal{B}_j]\!]^{\mathfrak{n}}$ y, al mismo tiempo, existe una función $f : 1..n \to 1..m$ tal que $[\![\mathcal{A}_i]\!]^{\mathfrak{n}} \preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}_{f(i)}]\!]^{\mathfrak{n}}$. Luego $\langle \mathcal{A}_i, \mathcal{B}_{f(i)}\rangle \in \mathcal{R}$, por lo que concluimos
  $$\langle \mathcal{A}, \mathcal{B}\rangle \;=\; \langle \oplus_{i\in 1..n}\mathcal{A}_i, \oplus_{j\in 1..m}\mathcal{B}_j\rangle \in \Phi_{\preceq_{\mathfrak{T}}}(\mathcal{R})$$

□

El lema anterior nos permite vincular la relación de subtipado para μ-types $\preceq_\mu$ y la nueva noción para $\mathfrak{T}^{\mathfrak{n}}$:

Proposición 6. *$A \preceq_\mu B$ si y sólo si $[\![[\![A]\!]^{\mathfrak{T}}]\!]^{\mathfrak{n}} \preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![[\![B]\!]^{\mathfrak{T}}]\!]^{\mathfrak{n}}$.*

Demostración. De Prop. 4 obtenemos $[\![A]\!]^{\mathfrak{T}} \preceq_{\mathfrak{T}} [\![B]\!]^{\mathfrak{T}}$ y concluimos apelando a Lem. 3. □

Este resultado establece que realizando las traducciones adecuadas puede utilizarse la relación $\preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}$ para computar $\preceq_\mu$. Se verá también que la formulación de la nueva relación como un caso particular de $\preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}}$ resulta conveniente a la hora de probar la corrección del algoritmo de verificación correspondiente.

3.2.1. Algoritmo de verificación

El algoritmo que computa la nueva relación de subtipado está inspirado en el presentado en [DCPR05], y tiene una complejidad de peor caso $O(NN'd)$, donde N y N' son el tamaño de los autómatas que describen a los tipos involucrados (considerando tanto estados como ejes) y d es una cota a la aridad de las uniones presentes en ambos. Para representar los autómatas se utiliza la técnica sugerida en [AC93], que permite comparar los mismos en $O(1)$ identificándolos por su posición en memoria.

Llamamos **válido** a un par $p \in \mathfrak{T}^{\mathfrak{n}} \times \mathfrak{T}^{\mathfrak{n}}$ si pertenece a $\preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}$. El algoritmo consiste en dos fases. El objetivo de la primera es construir un conjunto $U \subseteq \mathfrak{T}^{\mathfrak{n}} \times \mathfrak{T}^{\mathfrak{n}}$ que delimite un universo de pares de tipos que luego será refinado para obtener un conjunto compuesto únicamente por pares válidos. Como se puede observar en el pseudocódigo presentado en la Fig. 3.5, se parte del par inicial y se exploran los pares de subtérminos de ambos tipos, descomponiendo cada unión encontrada en todos sus componentes. Notar que, dado un par p, se podrían agregar aquí pares inválidos o innecesarios para probar la validez de p. El trabajo de descartar los mismos para probar la validez del par inicial se realizará en la etapa siguiente.

Observar que U puede interpretarse como un grafo dirigido en donde se agrega un eje entre los pares p y q si al momento de analizar p se determina la presencia de q en el resultado. En este caso diremos que p es **padre** de q. Dado que los términos pueden contener ciclos, un par puede ser agregado al conjunto más de una vez y por lo tanto tener más de un padre. Definimos $u(p)$, el **grado de entrada** de un par p, como su cantidad de padres.

En la segunda fase, cuyo pseudocódigo se muestra en la Fig. 3.6, se mantienen los siguientes tres conjuntos, que forman en todo momento una partición de U:

- W: pares cuya validez todavía debe ser determinada
- S: pares considerados válidos tomando como hipótesis $W \cup S$
- F: pares inválidos

El algoritmo toma repetidamente elementos de W y, en cada iteración, transfiere a S el par p seleccionado si la validez del mismo puede probarse asumiendo válidos sólo los pares no descartados hasta el momento. En caso contrario, se transfiere p a F y se retorna a W cualquier padre de p presente en S para ser eventualmente reconsiderado. Este proceso se denomina **invalidación**. Intuitivamente, S contiene en cada iteración elementos pertenecientes a la relación $\preceq^{W}_{\mathfrak{T}^{\mathfrak{n}}}$. El proceso termina cuando W es vacío, momento en el cual la validez de cualquier elemento de S puede justificarse utilizando sólo elementos del mismo conjunto. Llegado este punto, basta verificar en qué conjunto se encuentra el par inicial.

Corrección

La corrección del algoritmo está basada en que S puede considerarse un conjunto de pares válidos *asumiendo la validez de los pares en* W. Esta condición se mantiene a lo largo de la ejecución, a medida que se descartan pares inválidos con el objetivo de obtener una relación final compuesta únicamente por pares válidos. A continuación enunciamos este invariante y probamos su corrección:

$$
\begin{array}{lll}
\mathsf{buildUniverse}(p_0) & : & U = \emptyset \\
& & W = \{p_0\} \\
& & \texttt{while } W \neq \emptyset : \\
& & \quad p := \mathsf{takeOne}(W) \\
& & \quad \texttt{if } p \notin U \\
& & \quad\quad \mathsf{insert}(p,\ U) \\
& & \quad\quad \texttt{foreach } q \in \mathsf{children}(p) \\
& & \quad\quad\quad \mathsf{insert}(q,\ W) \\
& & \texttt{return } U \\
& & \\
\mathsf{children}(p) & : & \texttt{case } p \texttt{ of} \\
& & \quad \langle \mathcal{D} \mathbin{@} \mathcal{A}, \mathcal{D}' \mathbin{@} \mathcal{B} \rangle \to \\
& & \quad\quad \{\langle \mathcal{D}, \mathcal{D}' \rangle, \langle \mathcal{A}, \mathcal{B} \rangle\} \\
& & \quad \langle \mathcal{A} \supset \mathcal{B}, \mathcal{A}' \supset \mathcal{B}' \rangle \to \\
& & \quad\quad \{\langle \mathcal{A}', \mathcal{A} \rangle, \langle \mathcal{B}, \mathcal{B}' \rangle\} \\
& & \quad \langle \oplus_i^n \mathcal{A}_i, \oplus_j^m \mathcal{B}_j \rangle \to \\
& & \quad\quad \{\langle \mathcal{A}_i, \mathcal{B}_j \rangle \mid i \in 1..n,\ j \in 1..m\} \\
& & \quad \langle \oplus_i^n \mathcal{A}_i, \mathcal{B} \rangle,\ \mathcal{B} \neq \oplus \to \\
& & \quad\quad \{\langle \mathcal{A}_i, \mathcal{B} \rangle \mid i \in 1..n\} \\
& & \quad \langle \mathcal{A}, \oplus_j^m \mathcal{B}_j \rangle,\ \mathcal{A} \neq \oplus \to \\
& & \quad\quad \{\langle \mathcal{A}, \mathcal{B}_j \rangle \mid j \in 1..m\} \\
& & \quad \texttt{otherwise} \to \\
& & \quad\quad \emptyset
\end{array}
$$

Fig. 3.5: Pseudocódigo de la primera fase (construcción del universo U).

```
gfp(𝒜, ℬ)          :  W = buildUniverse(⟨𝒜, ℬ⟩)
                      S = ∅
                      F = ∅
                      while W ≠ ∅ :
                        p := takeOne(W)
                        if check(p, F)
                          then insert(p, S)
                          else invalidate(p, S, F, W)
                      return p ∈ S

check(p, F)        :  case p of
                        ⟨a, a⟩ →
                            true
                        ⟨𝒟 @ 𝒜, 𝒟′ @ ℬ⟩ →
                            ⟨𝒟, 𝒟′⟩ ∉ F and ⟨𝒜, ℬ⟩ ∉ F
                        ⟨𝒜 ⊃ ℬ, 𝒜′ ⊃ ℬ′⟩ →
                            ⟨𝒜′, 𝒜⟩ ∉ F and ⟨ℬ, ℬ′⟩ ∉ F
                        ⟨⊕_i^n 𝒜_i, ⊕_j^m ℬ_j⟩ →
                            ∀i.∃j. ⟨𝒜_i, ℬ_j⟩ ∉ F
                        ⟨⊕_i^n 𝒜_i, ℬ⟩,  ℬ ≠ ⊕ →
                            ∀i. ⟨𝒜_i, ℬ⟩ ∉ F
                        ⟨𝒜, ⊕_j^m ℬ_j⟩,  𝒜 ≠ ⊕ →
                            ∃k. ⟨𝒜, ℬ_k⟩ ∉ F
                        otherwise →
                            false

invalidate(p, S, F, W)  :  insert(p, F)
                           foreach q ∈ parents(p) ∩ S
                             move(q, S, W)
```

Fig. 3.6: Pseudocódigo de la segunda fase (refinamiento de la relación).

Proposición 7. *El algoritmo preserva el siguiente invariante:*

- $\langle W, S, F \rangle$ *es una partición de* U
- F *está compuesto únicamente por pares inválidos*
- $S \subseteq \Phi_{\preceq^W_{\mathfrak{T}^{\mathfrak{n}}}}(S)$

Demostración. Las tres condiciones son claramente válidas al comienzo de la segunda fase, donde $W = U$ y tanto S como F son vacíos. Por otro lado, a lo largo del algoritmo sólo se transfieren pares entre los tres conjuntos, preservando siempre la primera condición.

Observar que, en cada iteración, la decisión de invalidar o transferir a S el nodo p evaluado se toma analizando si existen en $S \cup W$ elementos suficientes para probar su validez. Probamos esto mediante un análisis de casos en la estructura de p:

- $p = \langle a, a \rangle$. Como indica (S-REFL-UP), la validez de p en este caso no depende de ningún otro par (*i.e.* su soporte es vacío). El algoritmo transfiere p a S directamente, sin verificar ningún otro elemento.

- $p = \langle \mathcal{D} @ \mathcal{A}, \mathcal{D}' @ \mathcal{A}' \rangle$. Por (S-COMP-UP) podemos ver que p es válido sólo si $\langle \mathcal{D}, \mathcal{D}' \rangle$ y $\langle \mathcal{A}, \mathcal{A}' \rangle$ lo son. Estos son precisamente los pares evaluados para decidir a dónde transferir p.

- $p = \langle \mathcal{A} \supset \mathcal{B}, \mathcal{A}' \supset \mathcal{B}' \rangle$. Este caso se corresponde con (S-FUNC-UP): p será movido a S sólo si $\langle \mathcal{A}', \mathcal{A} \rangle$ y $\langle \mathcal{B}, \mathcal{B}' \rangle$ no fueron aún descartados.

- $p = \langle \oplus_i^n \mathcal{A}_i, \oplus_j^m \mathcal{B}_j \rangle$. Aquí el algoritmo verifica si para cada $\mathcal{A}_i$ existe un $\mathcal{B}_j$ que permita seguir considerando a p válido. Estos son los pares $\langle \mathcal{A}_i, \mathcal{B}_{f(i)} \rangle$ mencionados en (S-UNION-UP).

- $p = \langle \oplus_i^n \mathcal{A}_i, \mathcal{B} \rangle$. Como indica (S-UNION-L-UP), se verifica que no haya un par $\langle \mathcal{A}_i, \mathcal{B} \rangle$ inválido.

- $p = \langle \mathcal{A}, \oplus_j^m \mathcal{B}_j \rangle$. De acuerdo a (S-UNION-R-UP), el algoritmo intenta encontrar el par $\langle \mathcal{A}, \mathcal{B}_j \rangle$ necesario para mantener p dentro del conjunto de pares condicionalmente válidos.

De no existir elementos suficientes en $S \cup W$ para probar la validez de p se determina que p es inválido. En este caso el algoritmo mueve el par a F, preservando la segunda condición.

Verificamos ahora la preservación de la última condición. Se distinguen con los subíndices 0 y 1 los estados de los conjuntos de la partición al inicio y final de una iteración respectivamente. Consideramos distintos casos según la decisión tomada sobre el par p evaluado:

- Si se decide mover p a S (*i.e.* todos los elementos necesarios para probar su validez se encuentran en $W_0 \cup S_0$), tenemos:

$$\begin{array}{rcl} W_1 & = & W_0 \setminus \{p\} \\ S_1 & = & S_0 \cup \{p\} \\ F_1 & = & F_0 \end{array}$$

Partimos de la hipótesis $S_0 \subseteq \Phi_{\preceq_{\mathfrak{T}^n}^{W_0}}(S_0)$ y operamos sobre ambos lados de la desigualdad para obtener

$$S_0 \cup \{p\} \quad \subseteq \quad \Phi_{\preceq_{\mathfrak{T}^n}^{W_0}}(S_0) \cup \{p\}$$

Observar que $S_1 \cup W_1 = S_0 \cup W_0$. Entonces, considerando la definición de $\Phi_{\preceq_{\mathfrak{T}^n}^{\mathcal{R}}}$ obtenemos $\Phi_{\preceq_{\mathfrak{T}^n}^{W_0}}(S_0) = \Phi_{\preceq_{\mathfrak{T}^n}^{W_1}}(S_1)$. Por lo tanto,

$$S_0 \cup \{p\} \quad \subseteq \quad \Phi_{\preceq_{\mathfrak{T}^n}^{W_1}}(S_1) \cup \{p\}$$

El conjunto a la izquierda de la desigualdad es precisamente S_1. Además, dado que los elementos necesarios para probar la validez de p se encuentran en $S_1 \cup W_1$, tenemos $p \in \Phi_{\preceq_{\mathfrak{T}^n}^{W_1}}(S_1)$. Concluimos entonces que

$$S_1 \subseteq \Phi_{\preceq_{\mathfrak{T}^n}^{W_1}}(S_1)$$

- Si se decide invalidar p, tenemos:

$$\begin{aligned} W_1 &= W_0 \setminus \{p\} \cup Q \\ S_1 &= S_0 \setminus Q \\ F_1 &= F_0 \cup \{p\} \end{aligned}$$

donde Q es el conjunto de padres de p pertenecientes a S_0. Para probar $S_1 \subseteq \Phi_{\preceq_{\mathfrak{T}^n}^{W_1}}(S_1)$ procedemos por el absurdo.

Supongamos que existe un $s \in S_1$ tal que $s \notin \Phi_{\preceq_{\mathfrak{T}^n}^{W_1}}(S_1)$. En particular s no puede tener soporte vacío. Además, como $S_1 \subseteq S_0$ tenemos $s \in S_0$.

Por otro lado, de $s \notin \Phi_{\preceq_{\mathfrak{T}^n}^{W_1}}(S_1)$ se deduce que existe un elemento necesario para probar su validez que no está en $S_1 \cup W_1$ pero sí está presente en $S_0 \cup W_0$, pues S_0 es $\Phi_{\preceq_{\mathfrak{T}^n}^{W_0}}(S_0)$-denso. En particular, el único elemento que difiere entre ambas uniones es p, lo que nos garantiza que s es padre de p. Dado que el algoritmo mueve a W_1 todos los padres de p presentes en S_0 debería valer $s \in W_1$, pero esto no puede ocurrir ya que $s \in S_1$ y los conjuntos W_1, S_1, F_1 forman una partición.

El absurdo proviene de suponer que existe tal s, por lo que concluimos

$$S_1 \subseteq \Phi_{\preceq_{\mathfrak{T}^n}^{W_1}}(S_1)$$

□

Al finalizar el ciclo principal sabemos que W es vacío, por lo que en este momento vale $S \subseteq \Phi_{\preceq_{\mathfrak{T}^n}^{\emptyset}}(S)$. Apelando al principio de coinducción obtenemos finalmente $S \subseteq \preceq_{\mathfrak{T}^n}^{\emptyset}$ (*i.e.* todos los pares en S son válidos). Solamente basta, entonces, verificar si el par original se encuentra en S o en F.

Análisis de complejidad

Para justificar la complejidad del algoritmo asumiremos que pueden realizarse operaciones sobre conjuntos en $O(1)$. Dado que siempre se trabaja sobre un conjunto acotado de elementos, esta garantía puede cumplirse numerando dichos elementos e indexándolos en un arreglo. Más adelante se discutirán alternativas más convenientes en la práctica. Con esta consideración, el costo de la primera fase es el de recorrer los subtérminos de los tipos a verificar, agregando en cada paso el par de subtérminos actual al resultado. Si llamamos N y N' al tamaño de los tipos a verificar, el tamaño de U (y el costo de su construcción) se acota por $O(NN')$. Como vemos a continuación, la complejidad de la verificación de subtipado está gobernada por la segunda fase.

Para acotar la cantidad de iteraciones observamos que el tamaño inicial de W es la cantidad de pares en U, y sólo se agregan pares a W al invalidar elementos. Dado que nunca se quitan elementos de F, un par puede ser invalidado a lo sumo una única vez. Por otro lado, al invalidar un par p se transfieren a lo sumo $u(p)$ nodos de S a W. De esto se deduce que la cantidad total de iteraciones es

$$\sum_{p \in U} 1 + \sum_{p \in U} u(p) \quad = \quad \sum_{p \in U}(1 + u(p)) \quad = \quad size(U)$$

Asumimos para analizar el costo de cada iteración que dado un par $p \in U$ es posible obtener eficientemente sus padres. Esto se logra en la implementación representando el grafo U con un conjunto que contiene un nodo por cada par de tipos del grafo, en los cuales se almacena un puntero hacia cada padre. La representación elegida resulta conveniente a la hora de invalidar un nodo: para obtener los padres de un elemento presentes en S basta recorrer los ejes almacenados en el nodo, filtrando aquellos que no pertenecen a S.

La complejidad de procesar cada par p está determinada por dos factores: la cantidad de operaciones realizadas para decidir si es necesario invalidarlo y, en caso afirmativo, el costo de la misma invalidación. Para tomar la decisión se debe verificar que existan en $W \cup S$ elementos necesarios para probar la validez de p. Analizando el código de `check` se evidencia que todos los casos realizan una cantidad constante de operaciones exceptuando aquellos que involucran uniones. Allí es preciso verificar que para cada componente de la unión izquierda exista al menos uno de la derecha aún no invalidado. Esto puede resolverse en tiempo lineal en el tamaño de la unión (acotado por d), manteniendo una tabla por cada par de la forma $\langle \oplus_i^n A_i, \oplus_j^m B_i \rangle$, en donde se almacena para cada A_i la cantidad de pares $\langle A_i, B_j \rangle$ que aún no fueron invalidados (ver Fig. 3.7).

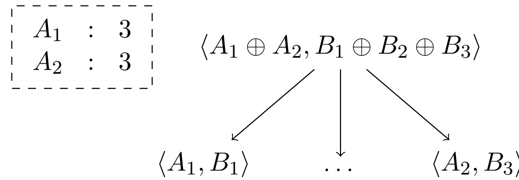


Fig. 3.7: Verificación de descendientes invalidados

Al momento de construcción del grafo, se asume que ningún par de componentes ha sido invalidado. De invalidarse luego el par $\langle A_1, B_2 \rangle$ se decrementa el contador asociado a A_1. Cuando más tarde se reconsidere el nodo padre, se decidirá invalidarlo si y sólo si los contadores asignados a todos los A_i son iguales a cero. En general, en el peor caso bastará con evaluar todos los contadores, por lo que la decisión puede tomarse con costo $O(d)$.

Observar por último que, en un análisis amortizado, el costo de invalidar un elemento puede verse como $O(1)$: por cada uno de los $u(p)$ pares agregados a W al invalidar p, se realizará una iteración adicional. Podemos, entonces, obtener una cota más fina para la ejecución entera del algoritmo si contamos el costo de agregar cada uno de estos elementos como parte de la iteración correspondiente. Por lo tanto, consideramos el costo amortizado de cada iteración como $O(d)$, lo que resulta en un costo global de $O(size(U)\ d)$ operaciones para el chequeo de subtipado, expresado como $O(NN'\ d)$ en término del tamaño de los autómatas de entrada.

Si consideramos n como la cantidad de constructores del tipo A, N es del orden de n^2 por ser el tamaño del autómata que lo representa. Con lo cual el orden de complejidad final del algoritmo puede expresarse como $O(n^2 n'^2 d)$, donde n y n' son los tamaños de A y B, respectivamente.

3.3. Equivalencia

Describimos en esta sección un desarrollo similar al presentado para la verificación de subtipado, presentando la noción de equivalencia $\simeq^{\emptyset}_{\mathfrak{T}^{\mathfrak{n}}}$ para árboles en $\mathfrak{T}^{\mathfrak{n}}$ y mostrando cómo puede utilizarse la misma para computar $\simeq_\mu$.

Definición 15. *Se define* $\simeq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}}$ *como* $\nu\Phi_{\simeq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}}}$, *donde* $\Phi_{\simeq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}}} : \wp(\mathfrak{T}^{\mathfrak{n}} \times \mathfrak{T}^{\mathfrak{n}}) \to \wp(\mathfrak{T}^{\mathfrak{n}} \times \mathfrak{T}^{\mathfrak{n}})$ *es la siguiente función:*

$$\begin{array}{rcl}
\Phi_{\simeq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}}}(\mathcal{S}) & = & \{\langle a, a\rangle \mid a \in \mathcal{V} \cup \mathcal{C}\} \\
& \cup & \{\langle \mathcal{D} \mathbin{@} \mathcal{A}, \mathcal{D}' \mathbin{@} \mathcal{A}'\rangle \mid \langle \mathcal{D}, \mathcal{D}'\rangle, \langle \mathcal{A}, \mathcal{A}'\rangle \in (\mathcal{S} \cup \mathcal{R})\} \\
& \cup & \{\langle \mathcal{A} \supset \mathcal{B}, \mathcal{A}' \supset \mathcal{B}'\rangle \mid \langle \mathcal{A}, \mathcal{A}'\rangle, \langle \mathcal{B}, \mathcal{B}'\rangle \in (\mathcal{S} \cup \mathcal{R})\} \\
& \cup & \{\langle \oplus_i^n \mathcal{A}_i, \oplus_j^m \mathcal{B}_j\rangle \mid \mathcal{A}_i, \mathcal{B}_j \neq \oplus, \\
& & \quad \exists f : 1..n \to 1..m \textit{ tal que } \langle \mathcal{A}_i, \mathcal{B}_{f(i)}\rangle \in (\mathcal{S} \cup \mathcal{R}), \\
& & \quad \exists g : 1..m \to 1..n \textit{ tal que } \langle \mathcal{A}_{g(j)}, \mathcal{B}_j\rangle \in (\mathcal{S} \cup \mathcal{R})\} \\
& \cup & \{\langle \oplus_i^n \mathcal{A}_i, \mathcal{B}\rangle \mid \mathcal{A}_i, \mathcal{B} \neq \oplus, \forall i \in 1..n. \langle \mathcal{A}_i, \mathcal{B}\rangle \in (\mathcal{S} \cup \mathcal{R})\} \\
& \cup & \{\langle \mathcal{A}, \oplus_j^m \mathcal{B}_j\rangle \mid \mathcal{A}, \mathcal{B}_j \neq \oplus, \forall k \in 1..m. \langle \mathcal{A}, \mathcal{B}_k\rangle \in (\mathcal{S} \cup \mathcal{R})\}
\end{array}$$

La relación entre $\simeq_{\mathfrak{T}}$ y $\simeq^{\emptyset}_{\mathfrak{T}^{\mathfrak{n}}}$ se establece en el siguiente lema, cuya demostración es análoga a la de su contraparte para subtipado (Lem. 3).

Lema 4. *Para todo* $\mathcal{A}, \mathcal{B} \in \mathfrak{T}$, $\mathcal{A} \simeq_{\mathfrak{T}} \mathcal{B}$ *si y sólo si* $[\![\mathcal{A}]\!]^{\mathfrak{n}} \simeq^{\emptyset}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}]\!]^{\mathfrak{n}}$.

Demostración. Ver Apéndice A. □

De este modo, vinculamos a través de las traducciones $[\![\cdot]\!]^{\mathfrak{T}}$ y $[\![\cdot]\!]^{\mathfrak{n}}$ la relación de equivalencia para μ-types $\simeq_\mu$ con $\simeq^{\emptyset}_{\mathfrak{T}^{\mathfrak{n}}}$ como se enuncia a continuación.

Proposición 8. $A \simeq_\mu B$ *si y sólo si* $[\![[\![A]\!]^{\mathfrak{T}}]\!]^{\mathfrak{n}} \simeq^{\emptyset}_{\mathfrak{T}^{\mathfrak{n}}} [\![[\![B]\!]^{\mathfrak{T}}]\!]^{\mathfrak{n}}$.

Demostración. De Prop. 5 obtenemos $[\![A]\!]^{\mathfrak{T}} \simeq_{\mathfrak{T}} [\![B]\!]^{\mathfrak{T}}$ y concluimos apelando a Lem. 4. □

Tal como ocurre con la relación de subtipado, el resultado anterior nos sugiere un método para computar la equivalencia de μ-types utilizando la representación basada en autómatas.

$$\frac{}{a \simeq^{\mathcal{R}}_{\mathfrak{T}^n} a}\ (\text{E-REFL-UP})$$

$$\frac{\mathcal{D}\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^n} \cup \mathcal{R})\ \mathcal{D}' \quad \mathcal{A}\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^n} \cup \mathcal{R})\ \mathcal{A}'}{\mathcal{D} @ \mathcal{A} \simeq^{\mathcal{R}}_{\mathfrak{T}^n} \mathcal{D}' @ \mathcal{A}'}\ (\text{E-COMP-UP}) \qquad \frac{\mathcal{A}\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^n} \cup \mathcal{R})\ \mathcal{A}' \quad \mathcal{B}\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^n} \cup \mathcal{R})\ \mathcal{B}'}{\mathcal{A} \supset \mathcal{B} \simeq^{\mathcal{R}}_{\mathfrak{T}^n} \mathcal{A}' \supset \mathcal{B}'}\ (\text{E-FUNC-UP})$$

$$\frac{\begin{array}{ll} \mathcal{A}_i\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^n} \cup \mathcal{R})\ \mathcal{B}_{f(i)} & f : 1..n \to 1..m \\ \mathcal{A}_{g(j)}\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^n} \cup \mathcal{R})\ \mathcal{B}_j & g : 1..m \to 1..n \end{array} \quad \mathcal{A}_i, \mathcal{B}_j \neq \oplus}{\bigoplus_i^n \mathcal{A}_i \simeq^{\mathcal{R}}_{\mathfrak{T}^n} \bigoplus_j^m \mathcal{B}_j}\ (\text{E-UNION-UP})$$

$$\frac{\mathcal{A}_i\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^n} \cup \mathcal{R})\ \mathcal{B} \text{ para todo } i \in 1..n \quad \mathcal{A}_i \neq \oplus \quad \mathcal{B} \neq \oplus}{\bigoplus_i^n \mathcal{A}_i \simeq^{\mathcal{R}}_{\mathfrak{T}^n} \mathcal{B}}\ (\text{E-UNION-L-UP})$$

$$\frac{\mathcal{A}\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^n} \cup \mathcal{R})\ \mathcal{B}_j \text{ para todo } j \in 1..m \quad \mathcal{A} \neq \oplus \quad \mathcal{B}_j \neq \oplus}{\mathcal{A} \simeq^{\mathcal{R}}_{\mathfrak{T}^n} \bigoplus_j^m \mathcal{B}_j}\ (\text{E-UNION-R-UP})$$

Fig. 3.8: Relación de equivalencia para árboles infinitos con uniones n-arias

3.3.1. Algoritmo de verificación

El algoritmo de verificación para $\simeq^{\varnothing}_{\mathfrak{T}^n}$ está basado en el mismo esquema que el presentado para subtipado. Observar que sólo es necesario redefinir las funciones children y check para adaptar dicho esquema a la relación de equivalencia *up-to* $\mathcal{R}$ definida.

En la primera fase, la única diferencia con respecto al algoritmo de subtipado es que al evaluar un nodo compuesto por dos tipos funcionales $\langle \mathcal{A} \supset \mathcal{B}, \mathcal{A}' \supset \mathcal{B}' \rangle$ se deben agregar como hijos los pares $\langle \mathcal{A}, \mathcal{A}' \rangle$ y $\langle \mathcal{B}, \mathcal{B}' \rangle$. Podría haberse evitado este cambio definiendo la regla en cuestión de modo que invierta el par $\langle \mathcal{A}, \mathcal{A}' \rangle$ en la hipótesis, del mismo modo que se hace en subtipado. Si bien esto es perfectamente válido, nos interesa resaltar que esquema algorítmico puede adaptarse a la necesidad de cambiar el universo inicial. Por otro lado, hacerlo de este modo evita la necesidad de demostrar la simetría de $\simeq^{\varnothing}_{\mathfrak{T}^n}$ para probar su correspondencia con $\simeq_{\mathfrak{T}}$ (ver regla (S-FUNC-T) en la Fig. 1.13). El pseudocódigo correspondiente se muestra en la Fig. 3.9.

Para la fase de refinamiento de la relación, la única diferencia radica en cómo se decide si invalidar un par evaluado o transferirlo al conjunto de nodos condicionalmente válidos. Como se observa en el pseudocódigo de la Fig. 3.10, en este caso los descendientes a verificar son aquellos determinados en las precondiciones de las reglas de $\simeq^{\mathcal{R}}_{\mathfrak{T}^n}$.

De forma similar al caso de subtipado, se preserva el invariante de que S es un conjunto de pares válidos asumiendo la validez de los elementos en W, llegando al finalizar la iteración a un estado final en el que vale $S \subseteq \simeq^{\varnothing}_{\mathfrak{T}^n}$.

Proposición 9. *El algoritmo preserva el siguiente invariante:*

- $\langle W, S, F \rangle$ *es una partición de* U
- F *está compuesto únicamente por pares inválidos*

$$
\begin{array}{lll}
\mathsf{children}(p) & : & \texttt{case}\ p\ \texttt{of} \\
& & \quad \langle \mathcal{D} \mathbin{@} \mathcal{A}, \mathcal{D}' \mathbin{@} \mathcal{B} \rangle \rightarrow \\
& & \qquad \{ \langle \mathcal{D}, \mathcal{D}' \rangle, \langle \mathcal{A}, \mathcal{B} \rangle \} \\
& & \quad \langle \mathcal{A} \supset \mathcal{B}, \mathcal{A}' \supset \mathcal{B}' \rangle \rightarrow \\
& & \qquad \{ \langle \mathcal{A}, \mathcal{A}' \rangle, \langle \mathcal{B}, \mathcal{B}' \rangle \} \\
& & \quad \langle \oplus_i^n \mathcal{A}_i, \oplus_j^m \mathcal{B}_j \rangle \rightarrow \\
& & \qquad \{ \langle \mathcal{A}_i, \mathcal{B}_j \rangle \mid i \in 1..n,\ j \in 1..m \} \\
& & \quad \langle \oplus_i^n \mathcal{A}_i, \mathcal{B} \rangle,\ \mathcal{B} \neq \oplus \rightarrow \\
& & \qquad \{ \langle \mathcal{A}_i, \mathcal{B} \rangle \mid i \in 1..n \} \\
& & \quad \langle \mathcal{A}, \oplus_j^m \mathcal{B}_j \rangle,\ \mathcal{A} \neq \oplus \rightarrow \\
& & \qquad \{ \langle \mathcal{A}, \mathcal{B}_j \rangle \mid j \in 1..m \} \\
& & \quad \texttt{otherwise} \rightarrow \\
& & \qquad \emptyset
\end{array}
$$

Fig. 3.9: Definición de descendientes en la fase de construcción del universo U.

$$
\begin{array}{lll}
\mathsf{check}(p, F) & : & \texttt{case}\ p\ \texttt{of} \\
& & \quad \langle a, a \rangle \rightarrow \\
& & \qquad \texttt{true} \\
& & \quad \langle \mathcal{D} \mathbin{@} \mathcal{A}, \mathcal{D}' \mathbin{@} \mathcal{B} \rangle \rightarrow \\
& & \qquad \langle \mathcal{D}, \mathcal{D}' \rangle \notin F \ \textsf{and}\ \langle \mathcal{A}, \mathcal{B} \rangle \notin F \\
& & \quad \langle \mathcal{A} \supset \mathcal{B}, \mathcal{A}' \supset \mathcal{B}' \rangle \rightarrow \\
& & \qquad \langle \mathcal{A}, \mathcal{A}' \rangle \notin F \ \textsf{and}\ \langle \mathcal{B}, \mathcal{B}' \rangle \notin F \\
& & \quad \langle \oplus_i^n \mathcal{A}_i, \oplus_j^m \mathcal{B}_j \rangle \rightarrow \\
& & \qquad \forall i. \exists j.\ \langle \mathcal{A}_i, \mathcal{B}_j \rangle \notin F \ \textsf{and}\ \forall j. \exists i.\ \langle \mathcal{A}_i, \mathcal{B}_j \rangle \notin F \\
& & \quad \langle \oplus_i^n \mathcal{A}_i, \mathcal{B} \rangle,\ \mathcal{B} \neq \oplus \rightarrow \\
& & \qquad \forall i.\ \langle \mathcal{A}_i, \mathcal{B} \rangle \notin F \\
& & \quad \langle \mathcal{A}, \oplus_j^m \mathcal{B}_j \rangle,\ \mathcal{A} \neq \oplus \rightarrow \\
& & \qquad \forall j.\ \langle \mathcal{A}, \mathcal{B}_j \rangle \notin F \\
& & \quad \texttt{otherwise} \rightarrow \\
& & \qquad \texttt{false}
\end{array}
$$

Fig. 3.10: Verificación de validez (condicional) de un par durante la fase de refinamiento.

- $S \subseteq \Phi_{\simeq^W_{\mathfrak{T}^n}}(S)$

Demostración. Análoga al caso de subtipado, ya que se utiliza el mismo esquema general. Partiendo de un universo con todos los pares relevantes, se refina la relación verificando en cada caso si algún elemento necesario para probar la validez del par evaluado ha sido invalidado. □

Observar también que la complejidad del algoritmo sigue estando gobernada por la cantidad de iteraciones efectuadas y las verificaciones necesarias al evaluar los descendientes de un par dado. En primer lugar, como se justificó previamente, un par puede reconsiderarse a lo sumo tantas veces como descendientes posea, por lo que la cantidad de iteraciones sigue siendo de orden $O(size(U))$ o, lo que es lo mismo, $O(NN')$.

Por otro lado, en cada iteración, el caso más costoso sigue siendo el del tratamiento de pares de la forma $\langle \oplus_i^n \mathcal{A}_i, \oplus_j^m \mathcal{B}_j \rangle$. Aquí es necesario no sólo verificar que cada $\mathcal{A}_i$ esté relacionado con algún $\mathcal{B}_j$ (como ocurre en el caso de subtipado) sino que además se debe validar la relación en sentido inverso. Esto se resuelve con dos pasadas lineales sobre los subtérminos involucrados, manteniendo entonces la complejidad total del algoritmo $O(NN'd)$.

3.4. Consideraciones de implementación

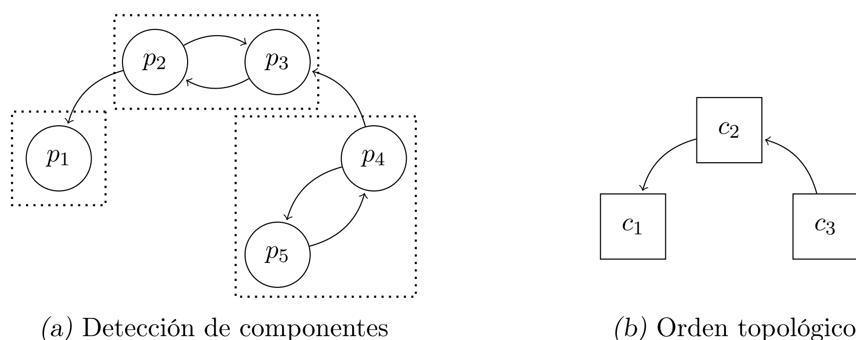
3.4.1. Representación de conjuntos y diccionarios

En ciertos algoritmos que trabajan sobre la representación cíclica de tipos se asumen representaciones eficientes de conjuntos y diccionarios[1]. En general estos usos responden a dos necesidades. Por un lado, cualquier recorrido sobre estas estructuras debe necesariamente llevar cuenta de qué nodos fueron ya visitados para determinar el momento de dar por terminado el recorrido. Además, suele ser necesario mantener cierta información asociada a cada nodo durante la ejecución de dichos algoritmos, para lo cual es conveniente utilizar diccionarios. En todos estos casos puede evitarse el uso de estas estructuras de datos externas almacenando dentro de los nodos los metadatos necesarios. Por ejemplo, podría modificarse la clase base `TypeNode` para incluir una variable de instancia `visited`, evitando así propagar durante el recorrido de la estructura un conjunto de nodos visitados.

Este enfoque, si bien aplicable, tiene sus desventajas. Incluir los datos almacenados por los algoritmos en la misma estructura de los nodos implica introducir estado en los mismos. El problema se manifiesta si, al momento de componer tipos, se construye un resultado con estado inconsistente. Para solucionar esto se podría, cuando sea necesario, clonar la estructura de nodos sobre la que se trabajó para asegurar que se manejan siempre autómatas sin historia. Tampoco es complicado definir funciones de limpieza para restaurar un autómata a su estado inicial tras cada corrida de un algoritmo, aunque se debe tener cuidado de utilizarlas debidamente tras cada operación realizada sobre las estructuras.

Con el objetivo de mantener la declaratividad de la solución se optó por mantener esta información en estructuras externas, utilizando representaciones de conjuntos y diccionarios basadas en funciones de hash que garantizan acceso en tiempo constante para el caso de uso promedio. Creemos que, en la práctica, esta solución de compromiso no penalizará determinantemente el desempeño del programa (pudiendo a veces beneficiarlo debido a los

[1] Los algoritmos en cuestión son las conversiones entre las representaciones de tipos `MUType` y `TypeNode` (y viceversa) y la averiguación del sort de un tipo (método `isDataType`).



(a) Detección de componentes *(b)* Orden topológico

Fig. 3.11: Optimización del orden de procesamiento de nodos

recorridos de limpieza evitados).

Por otro lado, la justificación de la complejidad de los algoritmos de subtipado y equivalencia depende fuertemente también de poder realizar operaciones sobre conjuntos de pares de tipos con costo de orden constante. Como se mencionó previamente, esto puede implementarse enumerando, al principio del algoritmo, cada par de subtérminos de los tipos trabajados e indexándolos en un arreglo. Sin embargo, se optó aquí nuevamente por favorecer la declaratividad de la solución, por lo que también en este caso se utilizaron conjuntos basados en funciones de hash.

3.4.2. Optimización del orden de selección de nodos

Observar que no se ha especificado hasta el momento el orden en que se deben tomar los elementos de W durante la segunda fase de los algoritmos de subtipado y equivalencia. Este no es un detalle menor: al invalidar un nodo se transfieren todos sus padres en S de nuevo al conjunto de trabajo, descartando efectivamente el trabajo realizado previamente sobre ellos. Para evitar suspender nodos que luego deban ser reconsiderados, resultaría deseable trabajar con los nodos descendientes primero, verificando las precondiciones de las reglas de forma ordenada. Sin embargo, dado que ante la presencia de tipos recursivos el grafo U puede contener ciclos, no es posible determinar en el caso general un orden que garantice que todo nodo será procesado antes que sus ancestros.

La implementación presentada incluye una optimización, sugerida en [DCPR05], que define un orden conveniente para reducir la cantidad de iteraciones incluso en presencia de ciclos. Terminada la primera fase se detectan las componentes fuertemente conexas[2] del grafo U. Dadas dos componentes unidas por un eje, se sabe que cualquier nodo de la componente destino es descendiente de todos los de la componente origen. Considerar, por ejemplo, el grafo de la Fig. 3.11. Al detectar las componentes fuertemente conexas podemos deducir que tanto p_2 como p_3 son descendientes de cualquier nodo de la componente conexa c_3. Por lo tanto, si bien no es posible definir un orden estricto entre esos dos nodos, sí tendrá sentido procesarlos antes que cualquier nodo de c_3.

La optimización consiste, entonces, en detectar las componentes fuertemente conexas de U y obtener un orden topológico inverso de las mismas (*i.e.* componentes destino primero). Luego, durante la ejecución de la segunda fase se toman los nodos de las componentes

[2] Una componente fuertemente conexa es un subgrafo en el cual todos los nodos son alcanzables entre sí.

ordenadas, de a una a la vez. El efecto de esta modificación es encapsular en las componentes fuertemente conexas los conjuntos de nodos para los cuales no es posible definir un orden correcto. Observar que al invalidar un nodo p sólo deberán reconsiderarse sus padres dentro de la misma componente fuertemente conexa: las componentes anteriores están compuestas sólo por descendientes de p, mientras que los nodos de las componentes posteriores no pueden haber sido evaluados aún. Notar también que en grafos acíclicos cada nodo constituye una componente fuertemente conexa unaria, por lo que en estos casos el orden resultante es el esperado: al invalidar nunca será necesario mover nodos de S a W.

El proceso de detección y ordenamiento de las componentes se implementó en este caso utilizando el algoritmo de Tarjan [DST80], cuyo costo de peor caso es $O(size(U))$, por lo que no se penaliza la complejidad del algoritmo original. Esta optimización es especialmente útil en grafos acíclicos, pero es de esperar que el rendimiento decaiga progresivamente en función de la cantidad de ciclos presentes, representando una mejora significativa desde el punto de vista práctico.

3.5. Discusión

En este capítulo buscamos resolver los problemas surgidos una vez obtenido el prototipo presentado en el capítulo anterior. Vimos que el orden de complejidad exponencial resultante de la poco conveniente representación utilizada podía reducirse a orden constante al adoptar una nueva representación que evitase tener que aplicar sustituciones al desdoblar un tipo. Con esto, funciones auxiliares como unfold bajan su complejidad a un simple chequeo lineal sobre el tamaño de entrada. Por otro lado, el algoritmo de subtipado desarrollado en esta segunda etapa nos permitió un análisis más preciso de su comportamiento asintótico, resultando en un orden $O(n^2n'^2d)$, siendo estos parámetros el tamaño de los tipos en cuestión y una cota a la aridad de sus uniones.

Retomemos ahora el análisis del algoritmo general de type checking (tc). Al trabajar con términos anotados y recibir el contexto como entrada del propio algoritmo, puede establecerse una relación lineal entre el tamaño del tipo resultante y dicha entrada. Es por esto que, en adelante, nos referiremos indistintamente al tamaño del tipo y la entrada.

Como se analizó previamente, tcp consiste en un recorrido lineal sobre el término, siendo sus iteraciones más costosas aquellas que trabajan sobre una aplicación o una abstracción. Como estos casos involucran llamados a subtipado, para simplificar su tratamiento asumiremos que las dos expresiones de tipo en cuestión tienen un tamaño similar, al que llamaremos n, refiriéndonos al orden de subtipado como $O(n^4d)$. Analizamos ahora los casos mencionados.

Aplicación: Se comienza realizando un chequeo lineal en el tipo para verificar si se trata de un datatype, en cuyo caso se retorna. De no ser así, es preciso realizar un segundo chequeo lineal (unfold) para luego realizar tantas llamadas a subtype como elementos contenga la unión de tipos funcionales. Esto nos da un orden de complejidad local de $O(n^4d^2)$.

Abstracción: Este caso comienza con tantos llamados a tcp (el algoritmo de chequeo de patrones) como ramas tiene la abstracción a analizar. Observar que tcp tiene una complejidad total lineal en el tamaño de su entrada, y éste es instanciado con los parámetros p_i y θ_i que provienen del término original. Por lo cual podemos considerar el costo de todos estos llamados como una única operación lineal en la entrada de tc.

Luego es preciso realizar una cantidad cuadrática (en la cantidad de ramas) de chequeos de compatibilidad. Analizamos previamente que la compatibilidad involucra, en el peor caso, un chequeo de subtipado. Si asumimos una cantidad lineal de ramas respecto de la entrada, obtenemos entonces un costo de complejidad local $O(n^6 d)$ para este caso.

Tras este análisis concluimos que la rama que gobierna la complejidad del algoritmo es aquella que analiza las abstracciones, resultando en una cota de complejidad $O(n^7 d)$ para el proceso completo de chequeo de tipos. Esto nos plantea la pregunta de qué tan representativa es esta cota en casos de uso real.

4. CONCLUSIONES

Con el objetivo de desarrollar un type checker eficiente para CAP se abordaron en este trabajo problemas de naturaleza técnica y práctica. Si bien las formulaciones dirigidas por sintaxis del sistema de tipos brindaban garantías sobre la tratabilidad del problema, no era evidente un método de verificación eficaz a partir de esta formalización. A efectos de la implementación resultó especialmente desafiante la combinación de tipos recursivos con operadores conmutativos, asociativos e idempotentes.

En la primera etapa del trabajo se desarrolló un algoritmo basado fuertemente en la presentación formal del sistema, enfoque que derivó en una primera implementación declarativa, pero cuyas limitaciones se evidenciaron al momento de analizar la complejidad de la solución. Esta versión sirvió, sin embargo, para identificar y establecer una base sobre la cual elaborar la solución a los problemas mencionados.

Las limitaciones de la primera implementación se atacaron empleando una representación conveniente para tipos recursivos y adoptando una técnica distinta para verificar la relación de subtipado, decisiones que tiñeron el segundo algoritmo de un carácter más imperativo. En este sentido resultó interesante la elección de Scala como lenguaje de implementación, que permitió encarar el cambio de paradigma preservando en gran medida la declaratividad de la implementación funcional desarrollada en la primera etapa.

Es interesante contrastar las técnicas utilizadas en ambas etapas a la hora de verificar la pertenencia a las relaciones coinductivas de subtipado y equivalencia. Si bien el algoritmo original tiene complejidad exponencial a causa de la representación de los datos utilizada, queda pendiente determinar si este problema podría haber sido resuelto únicamente adoptando la representación basada en autómatas. Como se mencionó oportunamente, para esto se deberá considerar el *backtracking* en el que se puede incurrir al procesar tipos con uniones. Sería interesante a futuro evaluar herramientas que permitan realizar un análisis riguroso sobre el comportamiento de esta adaptación del algoritmo y su relación con el segundo enfoque. A pesar de esto, creemos que el algoritmo utilizado en la segunda etapa goza de buenas propiedades desde el punto de vista práctico.

Más allá de la cota de complejidad obtenida para el peor caso, la optimización introducida en la elección de los pares de tipos a procesar mejora considerablemente las perspectivas del desempeño en la práctica. En [DCPR05] se sugiere otra optimización que podría contribuir en este sentido. Observar que en ciertos casos es posible declarar la invalidez de un par de tipos realizando solamente un análisis local. Por ejemplo, en el par $\langle \mathcal{A} \supset \mathcal{A}', \mathcal{D} \mathbin{@} \mathcal{B}' \rangle$ basta observar el constructor principal de ambos tipos para decidir $\mathcal{A} \supset \mathcal{A}' \not\preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} \mathcal{D} \mathbin{@} \mathcal{B}'$. Podría modificarse la fase de construcción del grafo U inicial para aprovechar este hecho y producir una falla temprana al analizar un par p con esta característica, siempre y cuando la validez del par p_0 inicial implique también la validez de p (*i.e.* cuando el camino entre p_0 y p no pase por pares con tipos unión). Se podría, por lo tanto, modificar el algoritmo para priorizar la exploración de caminos sin uniones. No resulta evidente, en principio, cómo cuantificar la mejora que podría aportar esta optimización. Continuando esta línea de trabajo, sería interesante elaborar un análisis empírico sobre el comportamiento del programa.

Por otra parte, actualmente se está trabajando en extender el sistema de tipos para capturar otras formas de polimorfismo. En vistas de contemplar estas extensiones a futuro, consideramos que el esquema de algoritmo utilizado en la implementación definitiva debería ser fácilmente adaptable a otras relaciones coinductivas.

Destacamos también que el algoritmo de type checking propuesto no se limita a verificar la validez de un juicio de tipado sino que construye un tipo asignable al término analizado. Aprovechando esto, una posible línea de trabajo futuro sería modificar el algoritmo actual para inferir el tipo más general asignable a un término sin anotaciones. Por último, otra extensión natural a este trabajo es la incorporación del desarrollo actual en el marco de un intérprete de expresiones del lenguaje.

Apéndice

A. DEMOSTRACIONES AUXILIARES

Demostración Lem. 4.
$\Rightarrow$) Probamos esta implicación mostrando que $\mathcal{R} \triangleq \{\langle \llbracket \mathcal{A} \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{B} \rrbracket^{\mathfrak{n}} \rangle \mid \mathcal{A} \simeq_{\mathfrak{T}} \mathcal{B}\}$ es $\Phi_{\simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}}$-denso. Se procede analizando la estructura de cualquier elemento posible de $\mathcal{R}$.

- $\langle \llbracket \mathcal{A} \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{B} \rrbracket^{\mathfrak{n}} \rangle = \langle a, a \rangle$. Entonces $\langle \llbracket \mathcal{A} \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{B} \rrbracket^{\mathfrak{n}} \rangle \in \Phi_{\simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}}(\mathcal{R})$.

- $\langle \llbracket \mathcal{A} \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{B} \rrbracket^{\mathfrak{n}} \rangle = \langle \mathcal{T}' \,@\, \mathcal{T}'', \mathcal{S}' \,@\, \mathcal{S}'' \rangle$. Notar que la traducción de un tipo en $\mathfrak{T}$ sólo puede tener un símbolo @ en la raíz si el tipo original también lo tiene. De aquí se infiere que $\mathcal{A}$ debe ser de la forma $\mathcal{D} \,@\, \mathcal{A}'$, donde $\llbracket \mathcal{D} \rrbracket^{\mathfrak{n}} = \mathcal{T}'$ y $\llbracket \mathcal{A}' \rrbracket^{\mathfrak{n}} = \mathcal{T}''$. De la misma manera sabemos que $\mathcal{B}$ es de la forma $\mathcal{D}' \,@\, \mathcal{B}'$, con $\llbracket \mathcal{D}' \rrbracket^{\mathfrak{n}} = \mathcal{S}'$ y $\llbracket \mathcal{B}' \rrbracket^{\mathfrak{n}} = \mathcal{S}''$.

  Dado que $\mathcal{A} \simeq_{\mathfrak{T}} \mathcal{B}$, sabemos que valen tanto $\mathcal{D} \simeq_{\mathfrak{T}} \mathcal{D}'$ como $\mathcal{A}' \simeq_{\mathfrak{T}} \mathcal{B}'$, lo que significa que $\{\langle \llbracket \mathcal{D} \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{D}' \rrbracket^{\mathfrak{n}} \rangle, \langle \llbracket \mathcal{A}' \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{B}' \rrbracket^{\mathfrak{n}} \rangle\} \subseteq \mathcal{R}$. Por lo tanto, $\langle \llbracket \mathcal{D} \rrbracket^{\mathfrak{n}} \,@\, \llbracket \mathcal{A}' \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{D}' \rrbracket^{\mathfrak{n}} \,@\, \llbracket \mathcal{B}' \rrbracket^{\mathfrak{n}} \rangle \in \Phi_{\simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}}(\mathcal{R})$. Usando nuevamente la definición de $\llbracket \cdot \rrbracket^{\mathfrak{n}}$, tenemos

$$\begin{array}{rcl} \langle \llbracket \mathcal{D} \rrbracket^{\mathfrak{n}} \,@\, \llbracket \mathcal{A}' \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{D}' \rrbracket^{\mathfrak{n}} \,@\, \llbracket \mathcal{B}' \rrbracket^{\mathfrak{n}} \rangle & = & \langle \llbracket \mathcal{D} \,@\, \mathcal{A}' \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{D}' \,@\, \mathcal{B}' \rrbracket^{\mathfrak{n}} \rangle \\ & = & \langle \llbracket \mathcal{A} \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{B} \rrbracket^{\mathfrak{n}} \rangle \in \Phi_{\simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}}(\mathcal{R}) \end{array}$$

- $\langle \llbracket \mathcal{A} \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{B} \rrbracket^{\mathfrak{n}} \rangle = \langle \mathcal{T}' \supset \mathcal{T}'', \mathcal{S}' \supset \mathcal{S}'' \rangle$. Este caso es similar al anterior. Primero inferimos que $\langle \mathcal{A}, \mathcal{B} \rangle$ es de la forma $\langle \mathcal{A}' \supset \mathcal{A}'', \mathcal{B}' \supset \mathcal{B}'' \rangle$, donde $\llbracket \mathcal{A}' \rrbracket^{\mathfrak{n}} = \mathcal{T}'$, $\llbracket \mathcal{A}'' \rrbracket^{\mathfrak{n}} = \mathcal{T}''$, $\llbracket \mathcal{B}' \rrbracket^{\mathfrak{n}} = \mathcal{S}'$, $\llbracket \mathcal{B}'' \rrbracket^{\mathfrak{n}} = \mathcal{S}''$. De $\mathcal{A} \simeq_{\mathfrak{T}} \mathcal{B}$ obtenemos $\{\langle \llbracket \mathcal{A}' \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{B}' \rrbracket^{\mathfrak{n}} \rangle, \langle \llbracket \mathcal{A}'' \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{B}'' \rrbracket^{\mathfrak{n}} \rangle\} \subseteq \mathcal{R}$, lo que significa que

$$\begin{array}{rcl} \langle \llbracket \mathcal{A}' \rrbracket^{\mathfrak{n}} \supset \llbracket \mathcal{A}'' \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{B}' \rrbracket^{\mathfrak{n}} \supset \llbracket \mathcal{B}'' \rrbracket^{\mathfrak{n}} \rangle & = & \langle \llbracket \mathcal{A}' \supset \mathcal{A}'' \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{B}' \supset \mathcal{B}'' \rrbracket^{\mathfrak{n}} \rangle \\ & = & \langle \llbracket \mathcal{A} \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{B} \rrbracket^{\mathfrak{n}} \rangle \in \Phi_{\preceq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}}(\mathcal{R}) \end{array}$$

- $\langle \llbracket \mathcal{A} \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{B} \rrbracket^{\mathfrak{n}} \rangle = \langle \oplus_i^n \mathcal{T}_i, \mathcal{S} \rangle$ con $\mathcal{S} \neq \oplus$. Un árbol de la forma $\oplus_i^n \mathcal{T}_i$ sólo puede ser resultado de traducir una unión maximal de n elementos, por lo que $\mathcal{A}$ debe ser $\oplus_{i \in 1..n} \mathcal{A}_i$, donde $\llbracket \mathcal{A}_i \rrbracket^{\mathfrak{n}} = \mathcal{T}_i$. Al mismo tiempo $\mathcal{S} \neq \oplus$ implica $\mathcal{B} \neq \oplus$, por lo que de $\mathcal{A} \simeq_{\mathfrak{T}} \mathcal{B}$ sabemos que para todo $i \in 1..n$ vale $\mathcal{A}_i \simeq_{\mathfrak{T}} \mathcal{B}$. Luego, $\langle \llbracket \mathcal{A}_i \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{B} \rrbracket^{\mathfrak{n}} \rangle \in \mathcal{R}$. Usando la definición de $\Phi_{\simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}}$ y la traducción obtenemos

$$\begin{array}{rcl} \langle \oplus_i^n \llbracket \mathcal{A}_i \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{B} \rrbracket^{\mathfrak{n}} \rangle & = & \langle \llbracket \oplus_{i \in 1..n} \mathcal{A}_i \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{B} \rrbracket^{\mathfrak{n}} \rangle \\ & = & \langle \llbracket \mathcal{A} \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{B} \rrbracket^{\mathfrak{n}} \rangle \in \Phi_{\simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}}(\mathcal{R}) \end{array}$$

- $\langle \llbracket \mathcal{A} \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{B} \rrbracket^{\mathfrak{n}} \rangle = \langle \mathcal{T}, \oplus_j^m \mathcal{S}_j \rangle$ con $\mathcal{T} \neq \oplus$. De forma similar al caso anterior, podemos asegurar que $\mathcal{A} \neq \oplus$ y $\mathcal{B} = \oplus_{j \in 1..m} \mathcal{B}_j$, donde $\llbracket \mathcal{B}_j \rrbracket^{\mathfrak{n}} = \mathcal{S}_j$. Dado que $\mathcal{A} \simeq_{\mathfrak{T}} \mathcal{B}$, sabemos que para todo $j \in 1..m$ vale $\mathcal{A} \simeq_{\mathfrak{T}} \mathcal{B}_j$ y por ende $\langle \llbracket \mathcal{A} \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{B}_j \rrbracket^{\mathfrak{n}} \rangle \in \mathcal{R}$. Sabemos entonces que

$$\begin{array}{rcl} \langle \llbracket \mathcal{A} \rrbracket^{\mathfrak{n}}, \oplus_j^m \llbracket \mathcal{B}_j \rrbracket^{\mathfrak{n}} \rangle & = & \langle \llbracket \mathcal{A} \rrbracket^{\mathfrak{n}}, \llbracket \oplus_{j \in 1..m} \mathcal{B}_j \rrbracket^{\mathfrak{n}} \rangle \\ & = & \langle \llbracket \mathcal{A} \rrbracket^{\mathfrak{n}}, \llbracket \mathcal{B} \rrbracket^{\mathfrak{n}} \rangle \in \Phi_{\simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}}(\mathcal{R}) \end{array}$$

- $\langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}} \rangle = \langle \oplus_i^n \mathcal{T}_i, \oplus_j^m \mathcal{S}_j \rangle$. Usamos el mismo argumento de los dos casos anteriores para inferir que se tienen uniones maximales $\mathcal{A} = \oplus_{i\in 1..n}\mathcal{A}_i$ y $\mathcal{B} = \oplus_{j\in 1..m}\mathcal{B}_j$, donde $[\![\mathcal{A}_i]\!]^{\mathfrak{n}} = \mathcal{T}_i$ y $[\![\mathcal{B}_j]\!]^{\mathfrak{n}} = \mathcal{S}_j$. De $\mathcal{A} \simeq_{\mathfrak{T}} \mathcal{B}$ sabemos que existen

$$\begin{array}{lll} f : 1..n \to 1..m & \text{tal que} & \mathcal{A}_i \simeq_{\mathfrak{T}} \mathcal{B}_{f(i)} \\ g : 1..m \to 1..n & \text{tal que} & \mathcal{A}_{g(j)} \simeq_{\mathfrak{T}} \mathcal{B}_j \end{array}$$

De esto se deduce que $\left\{\langle [\![\mathcal{A}_i]\!]^{\mathfrak{n}}, [\![\mathcal{B}_{f(i)}]\!]^{\mathfrak{n}} \rangle, \langle [\![\mathcal{A}_{g(j)}]\!]^{\mathfrak{n}}, [\![\mathcal{B}_j]\!]^{\mathfrak{n}} \rangle\right\} \subseteq \mathcal{R}$. De nuevo, por definición de $\Phi_{\simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}}$ y de la traducción obtenemos

$$\begin{array}{rcl} \langle \oplus_i^n [\![\mathcal{A}_i]\!]^{\mathfrak{n}}, \oplus_j^m [\![\mathcal{B}_j]\!]^{\mathfrak{n}} \rangle & = & \langle [\![\oplus_{i\in 1..n}\mathcal{A}_i]\!]^{\mathfrak{n}}, [\![\oplus_{j\in 1..m}\mathcal{B}_j]\!]^{\mathfrak{n}} \rangle \\ & = & \langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}} \rangle \in \Phi_{\simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}}}(\mathcal{R}) \end{array}$$

$\Leftarrow$) De modo similar, definimos $\mathcal{R} = \left\{\langle \mathcal{A}, \mathcal{B} \rangle \mid [\![\mathcal{A}]\!]^{\mathfrak{n}} \simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}]\!]^{\mathfrak{n}}\right\}$ y mostramos que es $\Phi_{\simeq_{\mathfrak{T}}}$-denso. Procedemos analizando la forma del cualquier posible par $\langle \mathcal{A}, \mathcal{B} \rangle \in R$:

- $\langle \mathcal{A}, \mathcal{B} \rangle = \langle a, a \rangle$. Entonces $\langle \mathcal{A}, \mathcal{B} \rangle \in \mathcal{R}$.
- $\langle \mathcal{A}, \mathcal{B} \rangle = \langle \mathcal{D} \mathbin{@} \mathcal{A}', \mathcal{D}' \mathbin{@} \mathcal{B}' \rangle$. La definición de $[\![\cdot]\!]^{\mathfrak{n}}$ nos dice que $\langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}} \rangle$ se define como $\langle [\![\mathcal{D}]\!]^{\mathfrak{n}} \mathbin{@} [\![\mathcal{A}']\!]^{\mathfrak{n}}, [\![\mathcal{D}']\!]^{\mathfrak{n}} \mathbin{@} [\![\mathcal{B}']\!]^{\mathfrak{n}} \rangle$. Además, dado $[\![\mathcal{A}]\!]^{\mathfrak{n}} \simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}]\!]^{\mathfrak{n}}$, deben valer tanto $[\![\mathcal{D}]\!]^{\mathfrak{n}} \simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{D}']\!]^{\mathfrak{n}}$ como $[\![\mathcal{A}']\!]^{\mathfrak{n}} \simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}']\!]^{\mathfrak{n}}$. Esto significa que $\{\langle \mathcal{D}, \mathcal{D}' \rangle, \langle \mathcal{A}', \mathcal{B}' \rangle\} \subseteq \mathcal{R}$, lo que implica

$$\langle \mathcal{A}, \mathcal{B} \rangle \;=\; \langle \mathcal{D}' \mathbin{@} \mathcal{A}', \mathcal{D}' \mathbin{@} \mathcal{B}' \rangle \in \Phi_{\simeq_{\mathfrak{T}}}(\mathcal{R})$$

- $\langle \mathcal{A}, \mathcal{B} \rangle = \langle \mathcal{A}' \supset \mathcal{A}'', \mathcal{B}' \supset \mathcal{B}'' \rangle$. Usando un razonamiento similar al anterior, $\langle [\![\mathcal{A}]\!]^{\mathfrak{n}}, [\![\mathcal{B}]\!]^{\mathfrak{n}} \rangle = \langle [\![\mathcal{A}']\!]^{\mathfrak{n}} \supset [\![\mathcal{A}'']\!]^{\mathfrak{n}}, [\![\mathcal{B}']\!]^{\mathfrak{n}} \supset [\![\mathcal{B}'']\!]^{\mathfrak{n}} \rangle$. Por otro lado, de $[\![\mathcal{A}]\!]^{\mathfrak{n}} \simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}]\!]^{\mathfrak{n}}$ obtenemos $[\![\mathcal{A}']\!]^{\mathfrak{n}} \simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}']\!]^{\mathfrak{n}}$ y $[\![\mathcal{A}'']\!]^{\mathfrak{n}} \simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}'']\!]^{\mathfrak{n}}$, por lo que $\{\langle \mathcal{A}', \mathcal{B}' \rangle, \langle \mathcal{A}'', \mathcal{B}'' \rangle\} \subseteq \mathcal{R}$. Entonces,

$$\langle \mathcal{A}, \mathcal{B} \rangle \;=\; \langle \mathcal{A}' \supset \mathcal{A}'', \mathcal{B}' \supset \mathcal{B}'' \rangle \in \Phi_{\simeq_{\mathfrak{T}}}(\mathcal{R})$$

- $\langle \mathcal{A}, \mathcal{B} \rangle = \langle \oplus_{i\in 1..n}\mathcal{A}_i, \oplus_{j\in 1..m}\mathcal{B}_j \rangle$, con $n + m > 2$ y $\mathcal{A}_i, \mathcal{B}_i \neq \oplus$.

  - Si $m = 1$ entonces $n > 1$, por lo que $[\![\mathcal{A}]\!]^{\mathfrak{n}} = \oplus_i^n [\![\mathcal{A}_i]\!]^{\mathfrak{n}}$. Dado que $[\![\mathcal{A}]\!]^{\mathfrak{n}} \simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}]\!]^{\mathfrak{n}}$, para todo $i \in 1..n$ tenemos $[\![\mathcal{A}_i]\!]^{\mathfrak{n}} \simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}]\!]^{\mathfrak{n}}$ y también $\langle \mathcal{A}_i, \mathcal{B} \rangle \in \mathcal{R}$. La definición de $\Phi_{\simeq_{\mathfrak{T}}}$nos dice que

  $$\langle \mathcal{A}, \mathcal{B} \rangle \;=\; \langle \oplus_{i\in 1..n}\mathcal{A}_i, \mathcal{B} \rangle \in \Phi_{\simeq_{\mathfrak{T}}}(\mathcal{R})$$

  - Si $n = 1$ entonces $m > 1$, por lo que $[\![\mathcal{B}]\!]^{\mathfrak{n}} = \oplus_j^m [\![\mathcal{B}_j]\!]^{\mathfrak{n}}$. $[\![\mathcal{A}]\!]^{\mathfrak{n}} \simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}]\!]^{\mathfrak{n}}$ significa que para todo $j \in 1..m$ vale $[\![\mathcal{A}]\!]^{\mathfrak{n}} \simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![\mathcal{B}_j]\!]^{\mathfrak{n}}$ y consecuentemente $\langle \mathcal{A}, \mathcal{B}_j \rangle \in \mathcal{R}$. Por lo tanto,

  $$\langle \mathcal{A}, \mathcal{B} \rangle \;=\; \langle \mathcal{A}, \oplus_{j\in 1..m}\mathcal{B}_j \rangle \in \Phi_{\simeq_{\mathfrak{T}}}(\mathcal{R})$$

  - Si $n > 1$ y $m > 1$ tanto $\mathcal{A}$ como $\mathcal{B}$ son uniones maximales de al menos dos elementos. Como en los casos anteriores, inferimos que $[\![\mathcal{A}]\!]^{\mathfrak{n}} = \oplus_i^n [\![\mathcal{A}_i]\!]^{\mathfrak{n}}$ y $[\![\mathcal{B}]\!]^{\mathfrak{n}} = \oplus_j^m [\![\mathcal{B}_j]\!]^{\mathfrak{n}}$, por lo que existen funciones

  $$\begin{array}{lll} f : 1..n \to 1..m & \text{tal que} & [\![\mathcal{A}_i]\!]^{\mathfrak{n}} \simeq_{\mathfrak{T}} [\![\mathcal{B}_{f(i)}]\!]^{\mathfrak{n}} \\ g : 1..m \to 1..n & \text{tal que} & [\![\mathcal{A}_{g(j)}]\!]^{\mathfrak{n}} \simeq_{\mathfrak{T}} [\![\mathcal{B}_j]\!]^{\mathfrak{n}} \end{array}$$

  por lo que $\left\{\langle \mathcal{A}_i, \mathcal{B}_{f(i)} \rangle, \langle \mathcal{A}_{g(j)}, \mathcal{B}_j \rangle\right\} \subseteq \mathcal{R}$. Concluimos entonces que

  $$\langle \mathcal{A}, \mathcal{B} \rangle \;=\; \langle \oplus_{i\in 1..n}\mathcal{A}_i, \oplus_{j\in 1..m}\mathcal{B}_j \rangle \in \Phi_{\simeq_{\mathfrak{T}}}(\mathcal{R})$$

$\square$

Bibliografía

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Trans. Program. Lang. Syst.*, 15(4):575–631, 1993.

[BDS13] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Cambridge University Press, New York, NY, USA, 2013.

[Cou83] Bruno Courcelle. Fundamental properties of infinite trees. *Theor. Comput. Sci.*, 25:95–169, 1983.

[DCPR05] Roberto Di Cosmo, François Pottier, and Didier Rémy. Subtyping recursive types modulo associative commutative products. In *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications*, TLCA'05, pages 179–193, Berlin, Heidelberg, 2005. Springer-Verlag.

[DST80] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, October 1980.

[EVB15] Juan Edi, Andrés Viso, and Eduardo Bonelli. Efficient type checking for path polymorphism. 2015. Draft.

[GD03] Andrzej Granas and James Dugundji. *Fixed point theory*. Springer monographs in mathematics. Springer, New York, Berlin, Heidelberg, 2003.

[JK06] C. Barry Jay and Delia Kesner. Pure pattern calculus. In Peter Sestoft, editor, *ESOP*, volume 3924 of *LNCS*, pages 100–114. Springer, 2006.

[JK09] C. Barry Jay and Delia Kesner. First-class patterns. *J. Funct. Program.*, 19(2):191–225, 2009.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[VBAR15] Andrés Viso, Eduardo Bonelli, and Mauricio Ayala-Rincón. Type soundness for path polymorphism. In *Proceedings of the 10th Workshop on Logical and Semantic Frameworks, with Applications*, ENTCS, 2015. To appear.